アジャイル開発モデルに基づくメタデータスキーマ設計手法と支援システム

アジャイル開発モデルに基づく
メタデータスキーマ設計手法と支援システム

筑波大学
図書館情報メディア研究科
2014 年 3 月
落合香織

目次

図目次

表目次

1. はじめに

様々なコミュニティによって Web 上に多種多様なメタデータが公開され、利用されている。特に最近では、誰でも利用、アクセス、再利用、再配布が可能なデータを Open Data[1]と呼び、国や自治体が持つ行政データを中心に Open Data 化が進められている。しかし多くのデータが公開されている一方で、その多くは計算機で処理しやすい形で公開されていない。また、データがどの様に記述されているのか、どの様に取り扱うべきかといった説明が十分に記されていない。そのため、実際にデータを利用する第三者にとって非常に利用しにくい状態である。Web の創始者と言える Tim Berners-Lee は、以下の様に Open Data として公開されるデータの特徴についての指標を五つ星で表現している[2]。図 1 はその指標を図示したものである[3]。

★1. （形式は問わず）メタデータをオープンライセンスで Web 上に公開すること
★2. メタデータを構造化データとして公開すること（例：表のスキャン画像よりも Excel）
★3. 非独占の形式を利用すること（例：Excel より CSV）
★4. 物事を示すために URI を利用すること
★5. メタデータのコンテキストを提供するために、他のメタデータへリンクすること

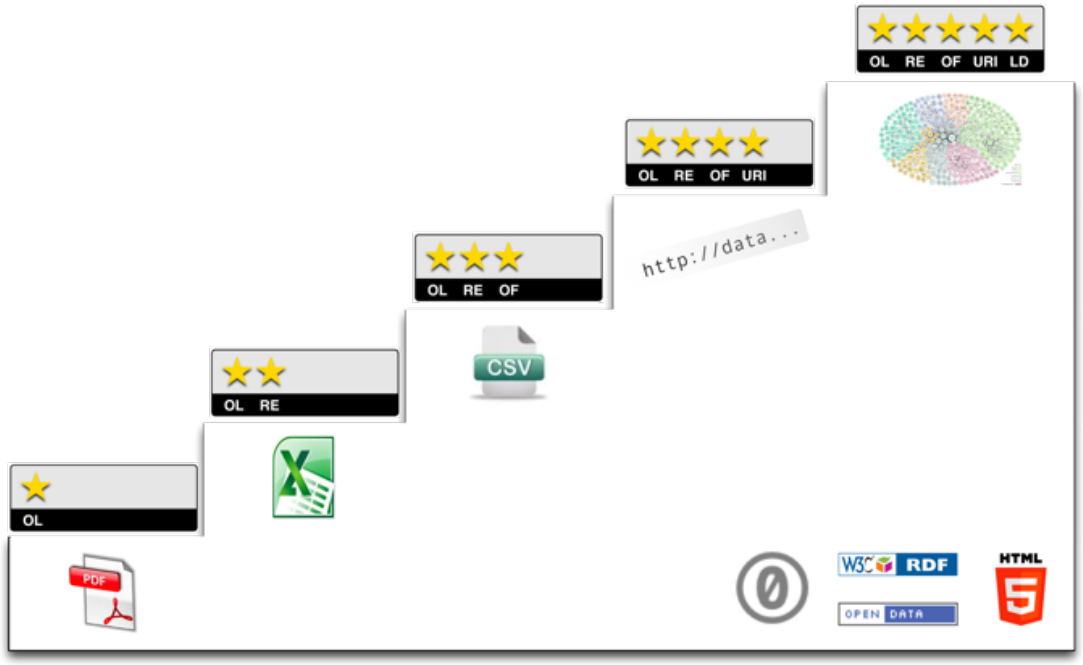


図 1　5 ★ Oepn Data：星が増えるほど相互利用性の高いデータとなる

ここで★4 の様に、物事を全て URI で示し、グラフ構造でデータを記述する表現を Resource Description Framework（RDF）と言う。また、★5 の様にデータを RDF で記述し、更に他のデータとリンクしている Open Data を Linked Open Data（LOD）と言う。RDF でデータを記述することで、計算機でデータを処理することが容易になり、データの再利用性が向上すると考えられる。更に他のデータとリンクさせることで、Web ページの様に、リンクから様々なデータを辿りながら、必要なデータを取得することが可能となる。

しかしながら、現在 Open Data として提供されているデータは★1 から★3 に該当するものがほとんどで、★4、★5 に該当するものはまだまだ少ない[28]。その理由の一つとして、RDF や LOD のデータを作成するために、まずどのように RDF でデータを記述するのかを詳細に設計する必要があり、それに手間がかかることが挙げられる。このデータ即ち、メタデータを記述するための決まりをメタデータスキーマと呼ぶ。メタデータ作成者の要求要件を満たしたメタデータスキーマを設計するためには、何度もメタデータの試作とメタデータスキーマの修正を繰り返す必要がある。加えて、相互利用性の高いメタデータスキーマを設計するためにはメタデータ設計者に専門的な知識や経験が求められる。そのため、メタデータスキーマの設計作業には非常に手間がかかるという問題がある。また、第三者にとってメタデータを使いやすい状態にするためには、メタデータと共にメタデータスキーマも公開される必要がある。

本研究では、既存のメタデータスキーマの設計プロセスについてまとめた上で、メタデータスキーマの設計手法の提案とその手法に対する支援システムの開発を行う。これにより、メタデータスキーマ設計者がより効率的に要求に沿ったメタデータスキーマを設計することを目指す。本稿では、2 章でメタデータの共有やメタデータスキーマについて、3 章でメタデータスキーマの設計プロセスと問題点について述べる。4 章では、ソフトウェア開発に用いられるアジャイル開発モデルに基づいてメタデータスキーマを設計するために、メタデータ作成ツールの開発をメタデータスキーマ設計に組み込んだ手法を提案する。5 章、6 章では、提案手法を実現するためにメタデータスキーマ設計を支援するシステムについて説明を行う。7 章以降では、支援システムに対する考察と本研究のまとめを述べる。

2. メタデータの共有

2.1. Linked Open Data

従来の Web は、HTML 文書がハイパーリンクで結び付けられて出来ている。人間は HTML 文書に書かれた情報を解釈し、利用する事が出来るが、計算機には難しい。しかし、計算機も情報の意味が解釈出来れば、人間の情報探索と利用を支援することが可能となる。計算機が情報の意味を解釈するために、従来の HTML 文書とそのリンクで構成される Web of Documents に加えて、計算機による解釈が可能なデータとそのリンクで構成される Web of Data を構築する必要がある。この様な考えは Semantic Web とも呼ばれる[4][5]。

この Web of Data の考えに沿って、データを公開・共有するための技術・取り組みを Linked Data と呼び、更にデータをオープンライセンスで公開したものを特に Linked Open Data（LOD）と呼ぶ。ここで、「オープンライセンス」とは、具体的には誰でも利用、アクセス、再利用、再配布が可能であることを指す。また LOD に限らず、オープンライセンスで公開されたデータを Open Data と呼ぶ。Linked Data は、具体的には以下の 4 つの原則を満たすものを指す[2]。

- ものの名前として URI を使うこと
- ものの名前の参照が HTTP URI で出来ること
- URI を見に行った時、RDF や SPARQL の様な標準技術によって、それに対する有用な情報を提供出来るようにすること
- より多くのものが発見出来るよう、データの中に他の URI へのリンクを入れること

Linked Data では、Resource Description Framework（RDF）[6]というモデルに従ってデータを記述する。RDF は Web 上でデータを共有するための標準モデルであり、図 2 の様な主語、述語、目的語の 3 つの組み合わせ（トリプル）でデータを記述する。例えば、図 2 では「http://mdlab.slis.tsukbua.ac.jp/」で識別されるものの「og:title（タイトル）」は「杉本・永森研究室」であるということを表現している。この様に RDF は非常にシンプルな考えであり、これらのトリプルを組み合わせることで、複雑な関係でも柔軟に記述することが可能である。RDF では、Web 上で識別出来るものを全てリソース（Resource）と呼び、その名前付けのために URI 参照を用いる。また、文字列や数値で表現するものをリテラル（Literal）と呼ぶ。RDF は、計算機が解釈しやすい形であることに加えて、リンクを使って別々の場所で公開されている複数のデータセットを結び付けることが出来る。つまり、LOD はまさに Web of Data の実現であると言える。また RDF では、SPARQL[26]という RDF クエリ言語を使ってデータの検索を行うことが出来る。LOD として公開され

ているデータセットは、SPARQL で検索可能な API として公開されていることが多い。この API のアクセス先となる URI を SPARQL Endpoint と呼ぶ。

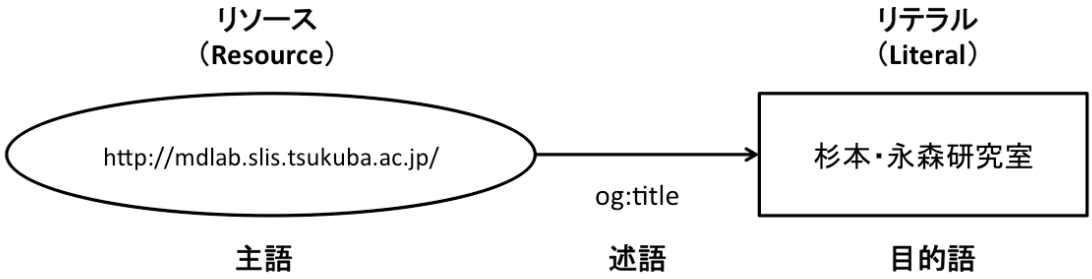


図 2　RDF グラフの例

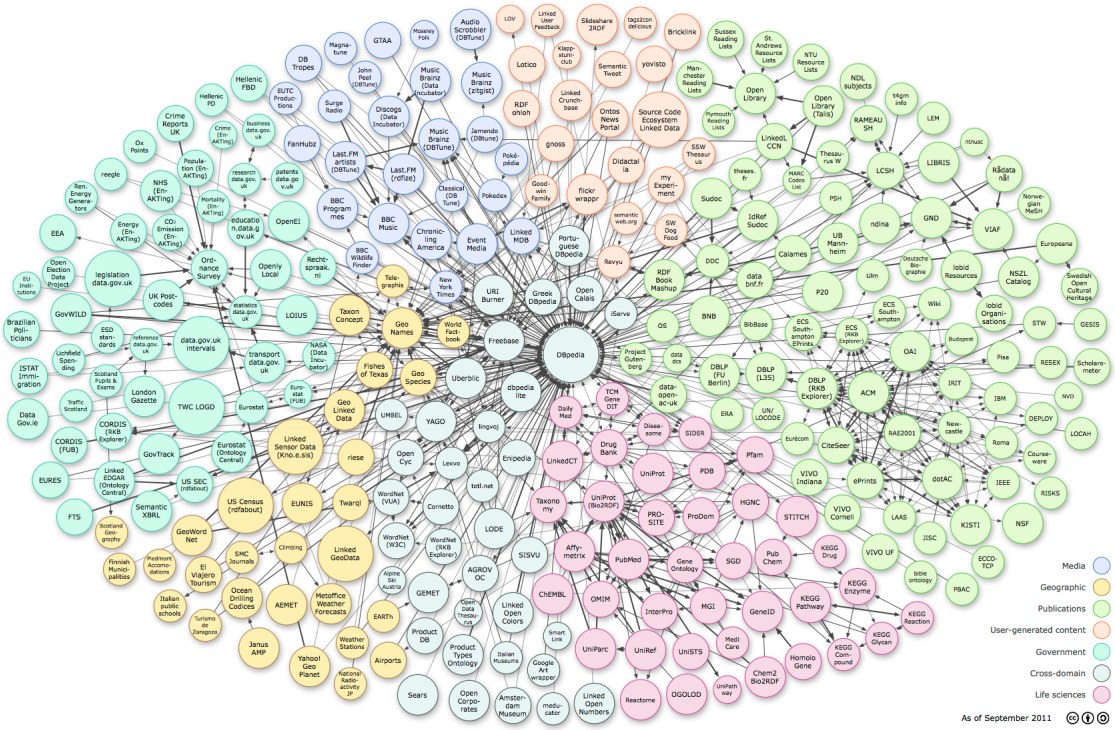


図 3　LOD クラウド

LOD の取り組みは広がってきており、295 個のデータセットが公開されている 2011 年の 9 月時点[8] から、現在も更に増え続けている。図 3 は 2011 年 9 月時点の LOD のリンクの広がりを表現したものである。

一方で、Open Data も急速に普及している。Open Data は地域やコミュニティ毎のポー

タルサイトで公開されることが多く、2014 年 1 月現在で、米国政府が提供する Data.gov[1]では 8 万件以上、英国政府が提供する Data.gov.uk[2]では 1 万 7000 件以上のデータセットが公開されている。日本でも 2013 年 12 月に、試行版として日本の各府省の保有データを公開する data.go.jp[3]が開設され、9000 件以上のデータセットが公開された。この様に、Open Data として公開されるデータセットの数と比べると、RDF や LOD の形で公開されるデータセットの割合はまだまだ小さい。その理由の一つに、RDF や LOD のデータを作成するために必要なメタデータスキーマの設計に手間がかかることが挙げられる。メタデータ作成者の要求要件を満たしたメタデータスキーマを設計するためには、何度もメタデータの試作とメタデータスキーマの修正を繰り返す必要がある。また、メタデータの相互利用性を踏まえたメタデータスキーマの設計には、RDF や LOD に関する専門知識や経験が求められる。本研究は、このメタデータスキーマの設計作業をより効率的に行えるようにすることを目的とする。次節では、まずメタデータスキーマについて説明し、3 章ではその設計プロセスと問題点について述べる。

2.2. メタデータスキーマ

メタデータとは、「Data about data」、あるもの・ことについてのデータを指す。例えば、ペットボトルのラベルは、ペットボトルの中に入っている液体について説明しているデータであり、まさにメタデータと言える。このことから、データという言葉はメタデータよりも広義の意味を持つと考えられるが、本稿では、データとメタデータをほぼ同義で用いる。また本研究で扱うメタデータは原則として、RDF で記述することを前提とする。

メタデータスキーマとは、メタデータを記述する際の制約を定めたもので、メタデータの設計書と言えるものである。本研究では、Nagamori ら[9]とメタデータ基盤協議会[10]の考えに沿って、メタデータスキーマを主にメタデータ語彙定義、メタデータ記述規則の 2 つから構成されるものと捉える。本節では、2.2.1 項でメタデータ語彙定義について、2.2.2 項でメタデータ記述規則について詳しく説明する。

2.2.1. メタデータ語彙定義

メタデータ語彙定義とは、一般的には RDF スキーマ（RDFS）[11]を使って定義されるもので、RDF の記述に必要なプロパティやクラスを定義するためものである。また、RDFS 自身も RDF を用いて記述する。本研究では、このプロパティやクラスを総称してメタデー

[1] http://www.data.gov/
[2] http://data.gov.uk/
[3] http://www.data.go.jp/

タ語彙と呼ぶ。

ここでプロパティとはリソース間の関係や主語リソースの特性を表すもので、RDF の述語に用いられる。クラスとはリソースをグループ化するタイプ（型）、あるいはカテゴリと捉えられるものである。図 4 の RDF グラフでは、「http://ci.nii.ac.jp/nrid/9000240207775#me」で識別されるリソースは「落合 香織」という「foaf:name（名前）」を持ち、「foaf:Person（人）」というグループに属することを表している。図 4 の「foaf:name」、「rdf:type」がプロパティ、「foaf:Person」がクラスである。RDF では、プロパティとクラスを全て URI で表現する。「foaf:name」や「foaf:Person」は名前空間接頭辞を用いた略称であり、「http://xmlns.com/foaf/0.1/name」、「http://xmlns.com/foaf/0.1/Person」と表記することもできる。名前空間接頭辞「foaf:」で省略される URI（この場合、「http://xmlns.com/foaf/0.1/」）の部分は、名前空間 URI と呼ぶ。

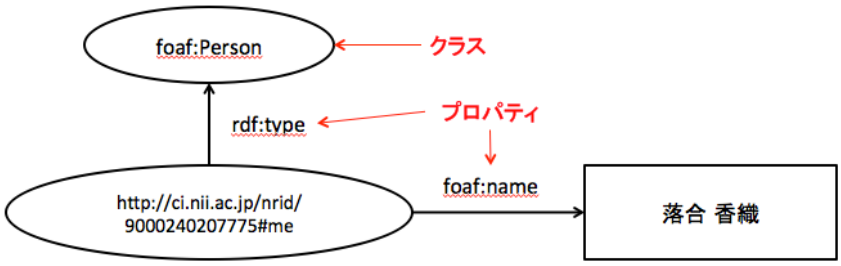


図 4　クラスとプロパティ

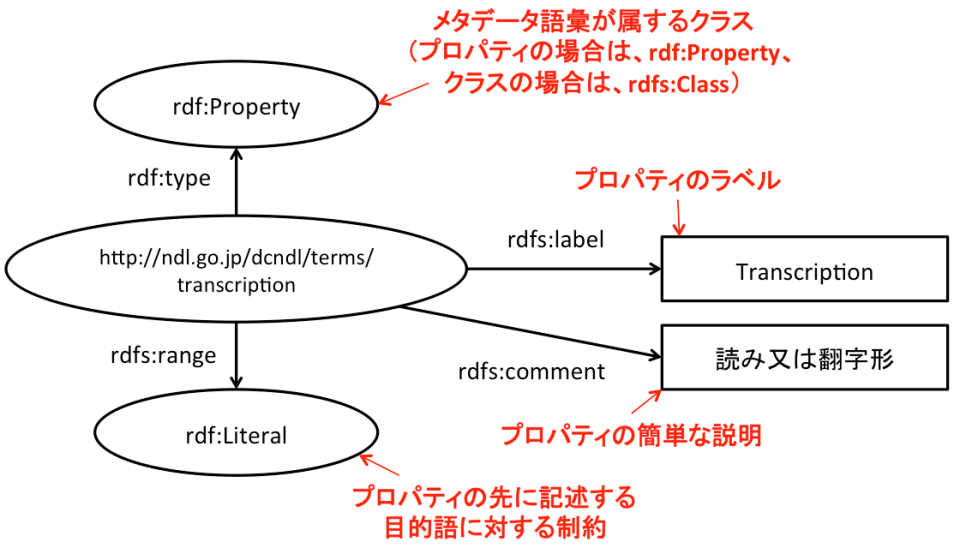


図 5　メタデータ語彙定義の例

図 5 はメタデータ語彙定義の例であり、「http://ndl.go.jp/dcndl/terms/transcription」は、ラベルが「Transcription」で、「読み又は翻字形」を記述するためのプロパティであるということを定義している。また、このプロパティを述語に用いた場合、目的語はリテラルでなければならないという制約を定義している。この様に、メタデータ語彙定義では、プロパティやクラスとなる URI 自身がどういうものであるかを説明すると同時に、簡単な制約を付けることも可能となっている。

メタデータ語彙定義は、誰でも作ることが可能で、様々な人やコミュニティによって定義され Web 上で公開されている。有名なものでは Dublin Core Metadata Initiative（DCMI）が定義した、ドキュメント等のメタデータに対して汎用的に用いることが出来る DCMI Metadata Terms[4]がある。また、Dan Brickley らが提案する、人や組織についてのメタデータを表現するために用いられる FOAF[5]がある。最近では、Google や Yahoo 等が協力して提供する Schema.org[6]や、Facebook が提供する The Open Graph Protocol[7]など企業が定義しているメタデータ語彙も広く普及している。多くの場合、メタデータ語彙の名前空間 URI にアクセスするとメタデータ語彙定義に辿り付ける。また、既に定義されているメタデータ語彙をそのまま利用するだけでなく、誰でも新たに定義を加えて機能を拡張することも出来る。

2.2.2. メタデータ記述規則

メタデータ記述規則とは、あるメタデータを記述する際に定める、そのメタデータを記述するための規則、制約の定義である。この規則は、記述に用いる項目とその値のためのメタデータ語彙の定義やそれぞれの記述項目は必須か任意か等と行った取り決め、即ちメタデータの構造的制約を定義する。加えて、メタデータを実現するための具体的実現形式、実際にメタデータを記述するために記述者に与えられるべきガイドラインも含まれる。このようにメタデータ記述規則は、

(1) メタデータ語彙
(2) 構造的制約（項目値記述の省略可能性や繰り返し条件などメタデータの構造的な制約）
(3) 具体的表現形式（システム上でのメタデータの具体的表現形式）
(4) メタデータ記述のガイドライン

---

4 http://dublincore.org/documents/dcmi-terms/
5 http://xmlns.com/foaf/spec/
6 http://schema.org/
7 http://ogp.me/

から成る[10]。ここで、(1) で述べたメタデータ語彙は 2.2.1 項で述べたメタデータ語彙定義で定義されたものを前提としている。メタデータ記述規則では、メタデータ語彙定義で定義された語彙から、記述したいメタデータに適切なものを選び取っている。そのため、メタデータ記述規則は図 6 の様にメタデータ語彙定義で定義されているメタデータ語彙を再利用する関係にある。

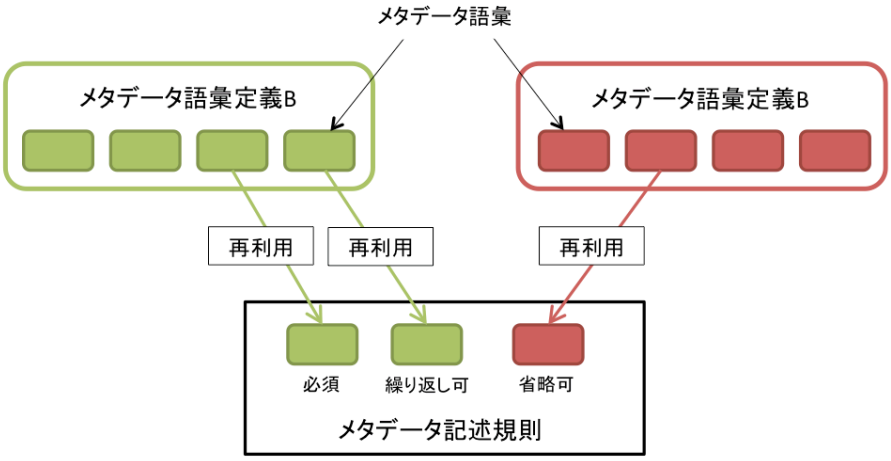


図 6　メタデータ語彙定義とメタデータ記述規則の関係

本研究では、メタデータ記述規則を形式的に表現するための書式として、DCMI が提案している Description Set Profile（DSP）[12] を採用している。DSP では、上記で述べた（1）～（2）を表現することが出来る。(3) と (4) の作成については本研究では支援の対象としない。また実際に用いる DSP は、メタデータ基盤協議会が提案する OWL-DSP[13] の形式で記述されたものを用いる。

DSP では、メタデータの構造に注目して制約を詳細に定義することが出来る。DSP は大きく Description Template と Statement Template の 2 つの要素で構成されている。図 7 は Description Template と Statement Template、それぞれで制約可能な範囲を図に示したものである。Statement Template はあるリソースに対する 1 つ 1 つのメタデータ記述項目についての制約を記述したものである。Description Template はリソース自身についての制約であり、Statement Template の集合を含んだものである。そのため、Description Template と Statement Template は常に 1:N（N は自然数）の関係となっている。

Description Template は主に以下の制約を記述することが出来る。ここでは、リソースに対する制約を記述する。

- Description Template の名前（ラベル）
- リソースが URI を持つかどうか（URI を持たないリソースを空白ノードと呼ぶ）
- リソースに付けるクラス（メタデータ語彙定義から選択）
- URI に対する制約（名前空間の指定など）
- Description Template が持つ Statement Template の集合

Statement Template は主に以下の制約を記述することが出来る。ここでは、プロパティや項目値についての制約を記述する。また、項目値には更に Description Template が続く場合もある。

- Statement Template の名前
- プロパティ（メタデータ語彙定義から選択）
- 項目値のタイプに対する制約（URI、リテラル、Description Template のどれか）
- 項目値の制約（リテラルの場合、文字列か数値か日付かなど、また統制語彙の様な特定の語彙の集合を定義することも出来る）
- Statement Template の最大出現回数（繰り返し不可ならば 1 など）
- Statement Template の最小出現回数（任意であれば 0、必須であれば 1 など）

これらの DSP の概念を、Web Ontology Language（OWL）[14]で表現可能にしたものが OWL-DSP である。OWL は、RDF を用いて表現することが出来るオントロジーを記述するための言語である。OWL は、2.2.1 項で説明した RDFS よりもクラスやプロパティの制約を詳細に記述論理に基づいて記述することが出来る。しかしながら、実際にメタデータを記述する際には、クラスやプロパティを中心とした制約だけでなく、本項で説明したメタデータの構造に対する詳細な制約の定義が求められる。OWL-DSP は OWL の機能を拡張し、それらの制約の表現を可能としている。

またメタデータ記述規則はメタデータを作成する時だけでなく、第三者がメタデータを利用する時、別のメタデータの作成に再利用する時にも求められると考えられる。そのためメタデータ記述規則を定義するだけでなく、メタデータと一緒に Web 上に公開することが必要である。次に 2.3 節では、メタデータ記述規則を共有する意義と現状について述べる。

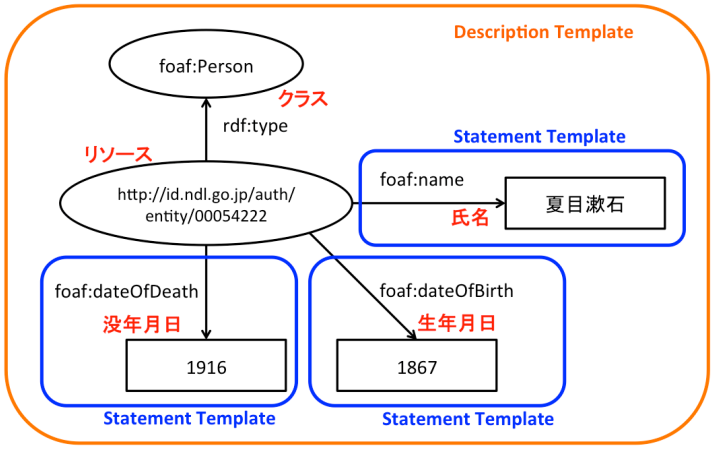


図 7　Description Template と Statement Template で記述可能な制約の範囲

2.3. メタデータ記述規則を共有するメリットと現状

多くの場合、RDF や LOD の形で公開されたメタデータの利用者は、2.1 節でも述べた RDF クエリ言語 SPARQL を用いて各データセットから必要な情報を取得する。しかし、SPARQL はクエリの中に RDF のグラフを記述し、そのグラフに該当する部分の情報を結果として返す。そのため、利用したいメタデータの構造的制約や使われているメタデータ語彙といった、メタデータ記述規則を利用者が把握する必要がある。しかしながら、データセット毎にメタデータ記述規則は異なるため、ユーザは実際にメタデータを確認しながらメタデータ記述規則を推測しなければならない。そのため、メタデータと共にメタデータ記述規則を第三者が利用出来る状態で Web 上に公開し共有することが求められる。更にメタデータ記述規則が計算機で処理することが可能となれば、ユーザはより簡単に様々な LOD データセットにアクセス可能になると考えられる。

しかし 2.2.1 項で述べた様に、メタデータ語彙定義が Web 上で公開される傾向にある一方で、メタデータ記述規則が Web 上に第三者がアクセス出来る状態で公開されることは少ない。この原因として、メタデータ語彙定義が RDFS や OWL で記述することが標準として提案されているのに対し、メタデータ記述規則の記述方法が定められていないことが考えられる。また、既に定義されているものを再利用出来るメタデータ語彙定義と比べ、メタデータ記述規則はメタデータを作成するコミュニティや要求要件が変わる度に細かい制約部分が異なってしまうため、メタデータスキーマ設計者が新たにドキュメントとしてまとめる必要があることも挙げられる。

本研究で提案する支援システムでは、メタデータ記述規則の設計を支援するだけでなく、メタデータ記述規則を計算機に処理可能なドキュメントである OWL-DSP のひな形として出力する。

3. メタデータスキーマの設計

本研究ではメタデータスキーマの設計の中でも、とりわけメタデータ記述規則の設計について着目し支援を行う。本章では、まず 3.1 節で既存のメタデータ記述規則の設計プロセスについて述べる。3.2 節では、3.1 節で紹介した文献と筆者の経験を踏まえた上で、メタデータ記述規則を設計するプロセスについてまとめる。そしてそれらを踏まえて、3.3 節でソフトウェア工学における設計プロセスの一つ、アジャイル開発モデルを紹介する。更に 3.4 節では、メタデータの相互利用性を向上するための指針について述べ、3.5 節で現在の設計プロセスの問題点について説明する。

3.1. Dublin Core Application Profile

DCMI は、メタデータ記述規則の設計プロセスとして『Guidelines for Dublin Core Application Profiles』[15]をまとめている。Dublin Core Application Profile（DCAP）は、DCMI が提案している、あるアプリケーションのためのメタデータ作成上の必要事項を一式定めたドキュメント、またその集まりを指す。DCAP を共有することで、メタデータの相互利用性を高められると考えられている。このガイドラインでは、DCAP のためのフレームワークである Singapore Framework [16]を紹介すると共に、DCAP を設計するプロセスについて説明している。Singapore Framework は以下の 5 つから構成されている。

- Functional Requirements（必須）…メタデータを用いるアプリケーションに対する要求要件
- Domain Model（必須）…メタデータの概念的なモデル
- Description Set Profile（DSP）（必須）…メタデータ記述項目の詳細な制約
- Usage Guidelines（任意）…DSP を補足するドキュメント
- Syntax Guidelines and Data Formats（任意）…メタデータの構文に関するガイドライン

これらは図 8 に示す様に、Functional Requirements から定義し、それを基に Domain Model を開発し、Domain Model を基に DSP を設計するといった依存関係となっている。実際に、『Guidelines for DCAPs』では以下のメタデータ記述規則設計プロセスを紹介している。

1. Functional Requirements を定義する（必須）
2. Domain Model を選択または開発する（必須）

3. メタデータ語彙を選択または定義する（必須）
4. DSP を使ってメタデータ記述項目を設計する（必須）
5. Usage Guidelines を作成する（任意）
6. Syntax Guidelines を作成する（任意）

本研究で提案する設計手順は、この『Guidelines for DCAPs』の手順を基にする。また本研究では、これらの手順の内、主に 2~4 までの設計プロセスを支援する。

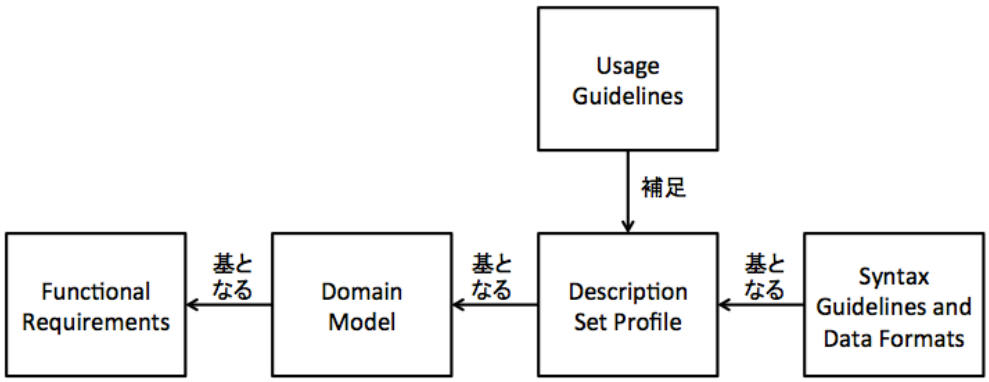


図 8　Singapore Framework

3.2. メタデータ記述規則の設計プロセス

3.1 節で述べた様に、メタデータ記述規則設計のためのガイドラインが提案されている一方で、実際のメタデータ記述規則の設計では、メタデータの作成や利用において発生した問題に合わせて、メタデータ記述規則の修正が行われる。例えば、値が記述不可能な項目がある、記述出来ても非常に手間がかかる、リソース同士を正しい関係で表現出来ていなかったなどというものである。その具体例を一つ紹介する。筆者は、著作権が切れた作品を電子化し Web 上で公開している電子図書館「青空文庫」の書誌や作家について記述するメタデータ[8]を作成した。図 9 は、そのときに設計したメタデータ記述規則の最初のバージョンの一部を図で示したものであり、ある本に対する著者と翻訳者の関係を表している。図 9 のバージョンでは、著者も翻訳者も「dcterms:creator」というプロパティで記述し、その目的語となる作家の URI に対してクラスとしてそれぞれ著者を表す「aozora:Author」と翻訳者を表す「aozora:Translator」と付けようと試みた。しかしこのメタデータ記述規則に沿ってデータをいくつか実際に作成し、利用してみると、作家によっては著者となる

[8] http://mdlab.slis.tsukuba.ac.jp/lodc2012/aozoralod/

場合も翻訳者である場合もあり、どちらのクラスも付けられ得ることが分かった。そのためこのメタデータ記述規則では、ある作品に対する著者と翻訳者を区別出来ず、メタデータの要求を満たすことが出来ないことが分かった。そこで図 10 の様に、クラスで区別するのではなく、著者には「dcterms:crator」、翻訳者には「bibo:translator」とうプロパティを使って区別する様に修正した。この様に、メタデータ記述規則の設計プロセスは、一度設計しただけでは十分に要件を満たすことが出来ず、メタデータの試作を通して問題を発見し、繰り返し修正を行う必要がある。

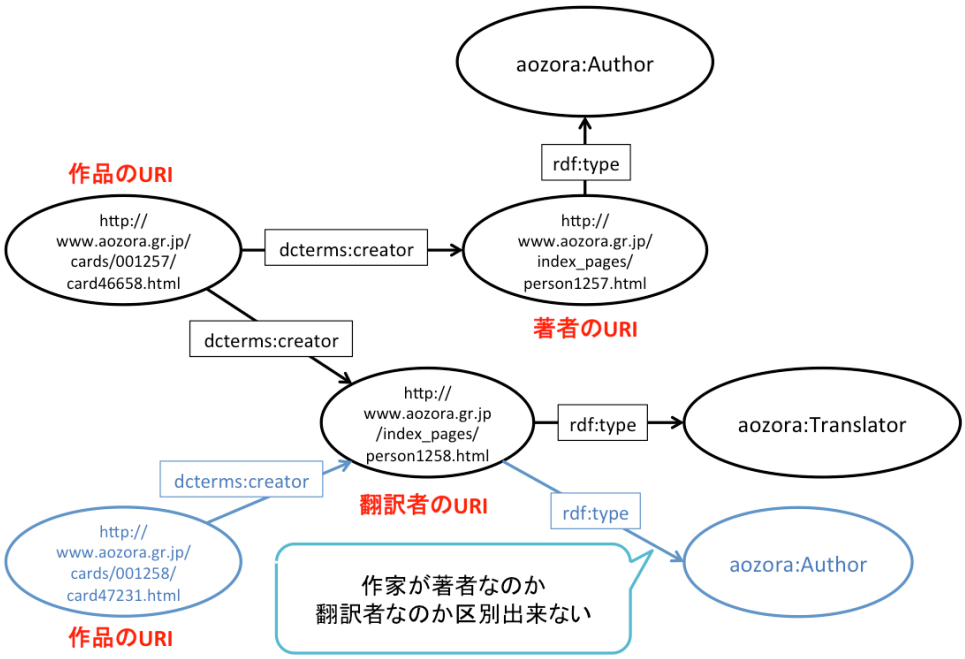


図 9　メタデータ記述規則の失敗例

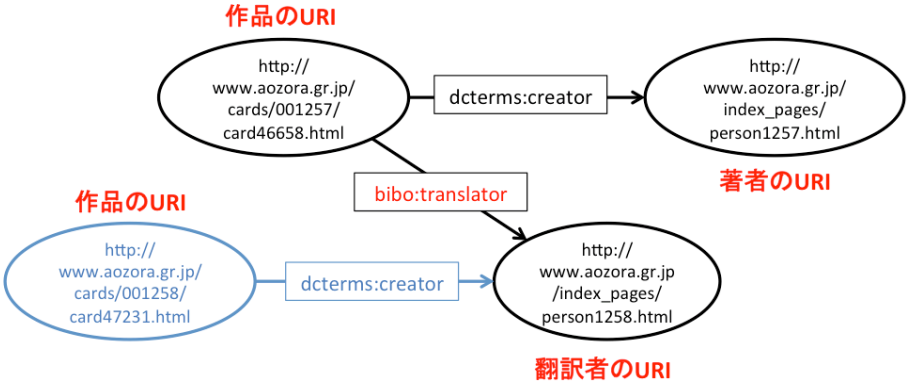


図 10　メタデータ記述規則の修正例

本研究では、上記で述べた筆者の経験と『Guidelines for DCAPs』を踏まえて、改めてメタデータ記述規則の設計プロセスを図 11 にまとめた。図 11 では、3.1 節の『Guidelines for DCAPs』の 2 から 4 のステップに加えて、メタデータの試作を通してメタデータ記述規則の評価を行なっている。更に、評価から改善点が見つかれば 2 から 4 のいずれかの行程に戻ってメタデータ記述規則の修正を行い、更にメタデータの試作を繰り返している。この様に、本研究ではメタデータ記述規則の設計プロセスは、メタデータの試作を通じて繰り返し行なわれるものであると捉えた。なお、ここでは作成するメタデータに対する要求分析・定義が既に定まっていることを前提としている。一方で、ソフトウェア工学の視点から見ると、同じ様に開発とフィードバックを繰り返すことで、顧客に価値のあるソフトウェアを提供することを目指すアジャイル開発モデルがある。そこで、本研究ではメタデータ記述規則の設計プロセスがアジャイル開発モデルに基づくものであると考えた。次節では、アジャイル開発モデルを紹介した上でメタデータ記述規則のプロセスとアジャイル開発モデルがどの様に共通するのかについて詳しく説明する。

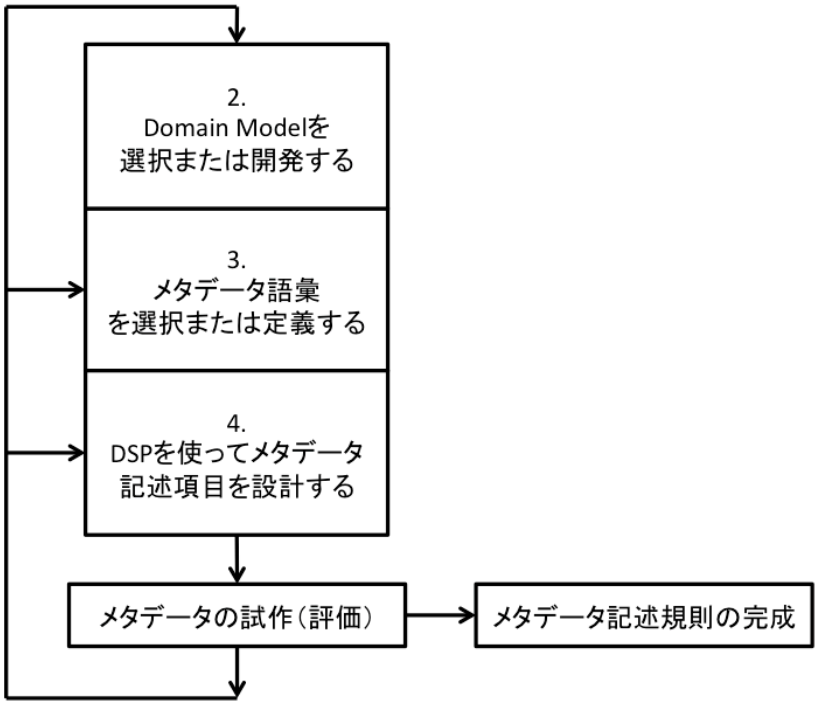


図 11　メタデータ記述規則の設計プロセス

3.3. アジャイル開発モデル

アジャイルソフトウェア開発とは、ソフトウェア工学において、仕様の変更などに柔軟に対応し、より迅速に顧客に価値のあるソフトウェアを提供すること目指す開発手法群の総称である[18][19]。本稿では、このアジャイルソフトウェア開発の考えに沿った開発モデルをアジャイル開発モデルと呼ぶ。アジャイル開発モデルでは、『アジャイルソフトウェア開発宣言』[20]がまとめられている。この宣言は、以下の 4 つの価値から成り立つ。

- プロセスやツールよりも個人と対話を
- 包括的なドキュメントよりも動くソフトウェアを
- 契約交渉よりも顧客との強調を
- 計画に従うことよりも、変化への対応を

このことからも、アジャイル開発モデルは仕様や計画を柔軟に変更し、常に動くソフトウェアを顧客に提供しながら、顧客が一番に求めるソフトウェアを開発していくものであることが分かる。アジャイル開発モデルのプロセスは、短い期間で動くソフトウェアを開発し、そのフィードバックを得るという行程を繰り返すという特徴を持つ。また要求を満たす条件をテストとして最初に考え、必ずソフトウェアがそれらを満たしているかを確認する。これは、要求要件を満たし、尚且つ確実に動くソフトウェアの開発を進めるために有効であると考えられる。

ここで、ソフトウェアをメタデータ記述規則に置き換えると、テストはメタデータ記述規則に沿って実際にメタデータを記述出来るかを確認する作業であり、これはメタデータの試作に等しい。そして、3.2 節でも述べた様にメタデータ記述規則の設計プロセスは、メタデータの試作、更にはその試作を通して出来たメタデータを利用することで得られるフィードバックを基に何度も繰り返される。このことから、メタデータ記述規則の設計プロセスは、アジャイル開発モデルに基づいたものであると考えられる。

3.4. メタデータ情報共有のためのガイドライン

3.1 節から 3.3 節までを通して、メタデータ記述規則の設計プロセスの流れと特徴について述べた。次に本節では、少し視点を変えてメタデータ記述規則の設計はどうあるべきかについて説明する。

メタデータ基盤協議会は、『メタデータ情報共有のためのガイドライン』[17]を提案している。このガイドラインは、メタデータの提供者、利用者双方を対象に、メタデータ記述規則の設計、メタデータの作成、利用、運用管理までメタデータの相互運用生、長期利用性を高めるための指針を示している。その中でも「スキーマの選択・設計と公開の指針」

では、メタデータの相互利用性を高めるために必要なこととして、出来るだけ広く普及した既存のメタデータ語彙利用すること、またそのメタデータ語彙の定義に従って利用することを述べている。つまり、メタデータスキーマ設計者は、メタデータ記述規則のためのメタデータ語彙を選択する際に、そのメタデータ語彙が広く普及しているかどうか、そのメタデータ語彙の定義に従ってデータを記述する事が出来るかを考える必要がある。

3.5. メタデータ記述規則の設計プロセスの問題点

3.2 節、3.3 節で説明した様に、メタデータ記述規則の設計プロセスはアジャイル開発モデルに基づきメタデータの試作を通じて何度も設計を繰り返すプロセスである。また、3.4 節では、そのプロセスにおいてメタデータスキーマ設計者がメタデータ記述規則を設計すする際に注意すべきことについて述べた。これらを踏まえて、本節ではメタデータ記述規則の設計プロセスの問題点について説明する。

まず一つ目の問題点は、メタデータの試作に手間がかかることである。メタデータの試作では、1 つか 2 つ程のメタデータを作成すればメタデータ記述規則の問題を発見出来る場合もある。しかし実際にメタデータを利用することで、メタデータ記述規則へのフィードバックを取得したい場合には、ある程度の量を試作する必要がある。ここで、メタデータはメタデータ作成者の手によって一つ一つ入力して作られることを想定すると、メタデータの入力・作成を支援するためのツールが求められると考えられる。しかし、一度メタデータを入力するためのツールを開発しても、メタデータ記述規則が変更されれば、そのツールを再度修正しなければならない。そのためツールの開発や修正の手間を省き、尚且つメタデータの試作を容易にするための支援が求められる。なお以降では、メタデータ作成者がメタデータを入力・作成を支援するためのツールをメタデータ作成ツールと呼ぶ。

二つ目の問題点は、メタデータの構造を設計する際やメタデータ語彙を選択する際に、メタデータスキーマ設計者は RDF や LOD に関する専門知識や経験が求められてしまうことである。その専門的な難しさは更に 2 つに分けられる。まず一つは、RDF のグラフ構造を設計する難しさである。3.2 節の失敗例も、グラフ構造に関するものであるが、図 12 の様な例もある。この例では、ある学生の学生氏名（name）とそのふりがな（name_kana）、姓（family_name）と名（given_name）を記述する。Pattern1 では学生（Student）から直接、学生氏名、ふりがな、姓、名のプロパティが伸びている。しかし場合によっては、学生氏名とふりがな、姓、名を、空白ノードを用いてまとめて記述する Pattern2 の様な構造が好まれることもある。この様なグラフ構造の表現パターンは、RDF や LOD に関する専門知識や経験を踏まえて考えられる。二つ目の専門的難しさはメタデータ語彙の選択である。メタデータ語彙定義が Web 上に多く公開されている一方で、その中から適切なメタ

データ語彙を探すことは非常に困難である。また、3.4 節でも述べた様に、メタデータの相互利用性を考えると出来るだけ広く普及したメタデータ語彙を用いることも重要である。最も意味が適切で、広く普及したメタデータ語彙を選ぶには、メタデータスキーマ設計者が RDF や LOD、メタデータ語彙に関する専門知識や経験がないと難しい。

以上 2 つの問題点を踏まえて本研究では、メタデータ作成ツールの開発を通じて、アジャイル開発モデルに基づいてメタデータ記述規則を設計する方法を提案する。

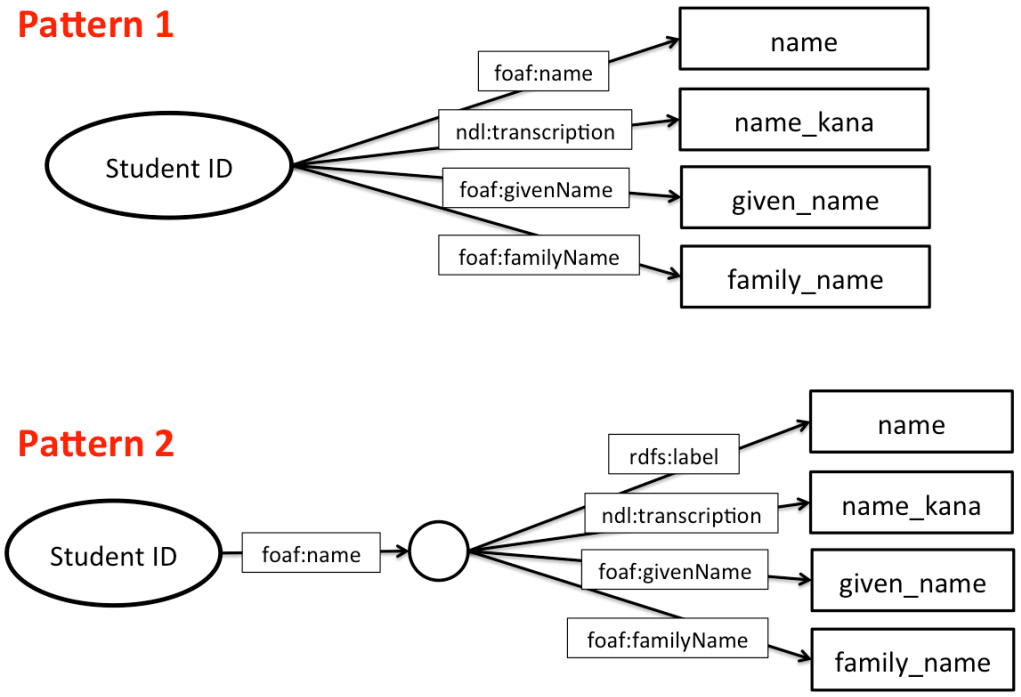


図 12　メタデータ記述規則のグラフ構造が異なる例

4. アジャイル開発モデルに基づくメタデータスキーマ設計手法

4.1. アジャイル開発モデルに基づくメタデータ記述規則の設計手法の提案

3 章で述べたように、本研究ではメタデータ記述規則の設計プロセスをアジャイル開発モデルに基づいたものとして考えた。本節では 3.5 節で述べた問題点を踏まえて、アジャイル開発モデルに基づいたメタデータ記述規則の設計をより行いやすくするために、メタデータ作成ツールの開発を通じて、メタデータ記述規則を設計する手法を提案する。なお、本研究のメタデータ作成ツールは、以下の 3 つの機能を持つものとする。

1. インタフェースとしてメタデータの値を入力するためのフォームを持つ
2. 入力したメタデータをリレーショナルデータベース（RDB）に保存する
3. 保存したメタデータを RDF として出力する

RDF で書かれたメタデータの保存には、RDF ストアなどを用いることも考えられるが、本研究ではこれから述べる 3 つのアイデアを実現するために、RDB を用いたメタデータ作成ツールを対象とした。

3.5 節では、メタデータ記述規則の設計プロセスには大きく 2 つの問題があると述べた。一つはメタデータの試作、またそれを容易にするためのメタデータ作成ツールの開発に手間がかかることである。もう一つは、RDF のグラフ構造の構築やメタデータ語彙の探索のために、RDF や LOD に関する専門知識や経験が必要となることである。これらの問題点に対して、本研究では以下の 3 つのアイデアを導き出した。

1. メタデータ記述規則の設計プロセスの一部を、メタデータ作成ツールの開発、とりわけデータベース（DB）スキーマの設計を通じて行う
2. メタデータ作成ツールの DB スキーマの情報を使って、メタデータ記述規則の設計プロセスの一部を支援する
3. メタデータ作成ツールの開発を、メタデータ記述規則の情報を使って支援する

アイデア 1 について説明する。メタデータ記述規則の設計プロセスの一部の行程で行うこと、例えば Domain Model の開発や必要なメタデータ記述項目の列挙、またそれらの最大出現回数や最小出現回数の決定などは、メタデータ作成ツールのための DB スキーマの設計で行う作業とほとんど変わらない。またメタデータ作成ツールから開発することで、項目欠如の有無や、適した値を記述出来ない項目の有無を先に確認しながらメタデータ記述規則を設計することが出来る。そこで本研究では、メタデータ作成ツールの DB スキー

マの設計を通じて、メタデータスキーマ設計者にメタデータ記述規則のおおまかな設計を行ってもらうことを考えた。

アイデア 2 について説明する。メタデータ記述規則の設計プロセスにおいて RDF のグラフ構造の構築やメタデータ語彙の探索は困難である。これは、RDF のグラフ構造を構築するためには、まず何についてのメタデータを記述したいかを考え、それらがどんな関係を持つかを決める必要があるためである。また、メタデータ語彙の探索には、リソースそのものを表すキーワードやその属性を表すキーワードが必要である。一方、メタデータ作成ツールのために設計された DB スキーマには、テーブル名やカラム名、テーブル同士の関連などといった情報が含まれている。そこで本研究ではメタデータ作成ツールの DB スキーマの情報を利用することで、メタデータスキーマ設計者に対して、グラフ構造の構築やメタデータ語彙の探索を支援することを考えた。これについて、詳しくは 5.1 節で述べる。

アイデア 3 について説明する。メタデータ記述規則の設計プロセスは、何度も繰り返し設計と試作を交互に行う。そのため、メタデータ記述規則が修正される度に、メタデータ作成ツールの修正が求められる。メタデータ作成ツールの開発では、まずどんなメタデータを記述したいか、またそれらがどんな関係を持つかを考え DB スキーマを決定する。そしてメタデータを入力するためのインタフェースや、RDF として出力するため仕組みを開発していく。ここでは、項目に対する制約や、RDF で表現するためのクラスやプロパティの情報が必要となる。一方、メタデータ記述規則にはメタデータの記述に用いるプロパティやクラスに加えて、構造的制約や項目値に対する制約が定義されている。そこで本研究では、メタデータ作成ツールの開発の手間を省くために、メタデータ記述規則の情報を利用してメタデータ作成ツールを簡単に開発するためのひな形を生成することを考えた。これについて、詳しくは 5.2 節で述べる。

本研究では、まず上記の 3 つのアイデアの実現のために、図 11 で示したメタデータ記述規則の設計プロセスを図 13 の様に 2 つのステップに分けた。ここでは、『Guidelines for DCAPs』の設計プロセスのうち、3 と 4 を更に細かく分割し、表 1 の様に割り当てている。Step1 はメタデータ記述規則のおおまかな設計部分、即ち Domain Model の開発やメタデータ記述項目の列挙、またそれらの最大出現回数、最小出現回数等の制約の決定といった作業を行う。Step2 は Step1 で決定したものを RDF へ適応させるための作業、即ち RDF のグラフ構造への適応やメタデータ語彙の探索を行う。Step2 まで終了すると、メタデータ記述規則を DSP として作成し、さらにメタデータの試作を行う。問題があれば、Step1 からやり直していく。

そして本研究では、この 3 つのアイデアに沿って、メタデータ作成ツールの開発行程を組み込んだメタデータ記述規則の設計手法を提案する。図 14 は提案手法を図に示したもの

である。本手法ではアイデア 1 に基づいて、まず Step1 としてメタデータ作成ツールの開発を行う。この時、メタデータスキーマ設計者はメタデータ作成ツールの DB スキーマの設計を行いながら、Domain Model の開発や必要なメタデータ記述項目の列挙、それらの制約の決定といったおおまかな設計を考えていく。そして、メタデータ作成ツールの DB スキーマの情報を基に RDF のグラフ構造の構築や、メタデータ語彙の探索を行っていく。Step1、Step2 を終えると、設計したメタデータ記述規則を DSP としてまとめ、その情報を基にメタデータ作成ツールを修正する。開発したメタデータ作成ツールを用いてメタデータの試作を行い、必要に応じてメタデータ作成ツールの DB スキーマを修正していく。そして、再び DB スキーマの情報を基に RDF のグラフ構造の構築やメタデータ語彙の探索を行い、メタデータ記述規則を DSP として作成する。

この提案手法を通して、メタデータスキーマ設計者は、メタデータ作成ツールの開発とメタデータ記述規則の設計を同時に行い、メタデータの試作を通じたメタデータ記述規則の設計をより効率的に行うことが可能となる。つまり、アジャイル開発モデルに基づいたメタデータ記述規則の設計プロセスをより効率的に行うことが可能となる。また、メタデータスキーマ設計者はメタデータ記述規則の設計に慣れていなくても、DB スキーマの設計を基にメタデータ記述規則を設計することが出来る様になる。これを実現するためには、「DB スキーマの情報を基に RDF のグラフ構造を構築し、メタデータ語彙を探索するための支援」と、「メタデータ記述規則の情報を基にメタデータ作成ツールのひな形を生成し、その開発を容易する支援」が必要である。5 章と 6 章では、これらを実際に行うための支援システムについて説明する。

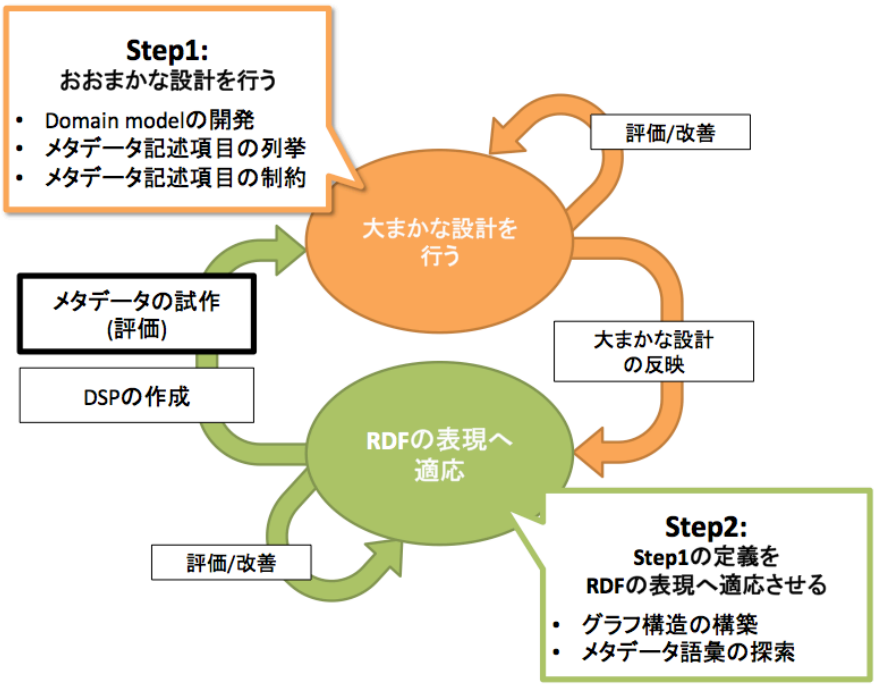


図 13　メタデータ記述規則の設計プロセスを 3 つのアイデアに適応させたもの

表 1　『Guidelines for DCAPs』と提案プロセスとの関係

Guidelines for DCAPs		本研究の設計プロセス	
2	Domain Model を選択または開発する	Step1	Domain Model を選択または開発する
3	メタデータ語彙を選択または定義する		メタデータ記述項目を列挙する
4	DSP を使ってメタデータ記述項目を設計する		メタデータ記述規則の制約を定義する
		Step2	グラフ構造の構築
3	メタデータ語彙を選択または定義する		メタデータ語彙を選択または定義する
4	DSP を使ってメタデータ記述項目を設計する		DSP の作成

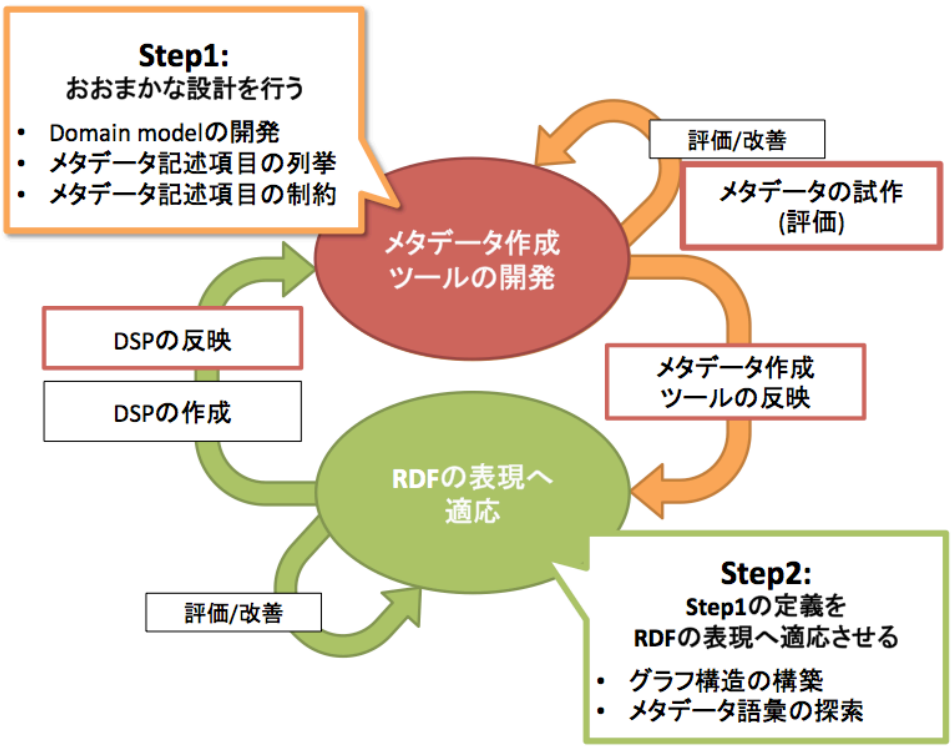


図 14　メタデータ作成ツールの開発を組み込んだメタデータ記述規則の設計手法

4.2. 関連研究

本研究の関連研究として、Malta らの Me4DCAP[21]がある。Me4DCAP（Method for the Development of DCAP）は、DCAP の反復的な設計手法を提案しているという点で本研究と共通する。しかしながら、Me4DCAP と本研究では、メタデータの試作に対する捉え方が異なる。Me4DCAP のメタデータの試作では、メタデータ記述項目の値を記述することが出来るか、また足りない記述項目がないかを確認し評価するために、ドキュメントにいくつか実際の値を記述する。本研究のメタデータの試作では Me4DCAP と同様に記述項目の確認を行うだけでなく、メタデータを試用することも想定し、メタデータ作成ツールを使って、ある程度量を含んだ RDF 形式のメタデータを作る。また、本研究では Domain Model の開発から DSP の設計までの部分に重点を置き、その行程を支援するためのシステム開発を行っているという点で異なる。

また、既存の RDB に保存された大量のデータを RDF に変換するための研究、ツールの開発が数多く行われている。その中には、DB スキーマの情報を基に RDF として記述するためのメタデータ記述規則を自動的に生成するものも含まれている[22][23][24]。しかし本研究は、DB スキーマの情報を基にメタデータ記述規則のひな形を生成するだけでなく、メタデータ記述規則から DB スキーマを用いた Web アプリケーションのひな形を生成しているという点で大きく異なる。また、これらの研究はどれも本研究で述べているメタデータ記述規則とは異なり、OWL を用いたオントロジーを出力しているという点で異なる。

5. メタデータ記述規則の設計支援システム

3 章ではメタデータ記述規則の設計プロセスの問題点について述べ、それを踏まえて 4 章ではアジャイル開発モデルに基づいて、メタデータ作成ツールの開発行程を組み込んだメタデータ記述規則の設計手法について説明した。本研究では、4 章で説明したメタデータ記述規則の設計手法をメタデータスキーマ設計者が実践するための支援システムを開発する。本章では主に、支援システムで行う支援内容とその手法について説明する。

本研究の支援システムでは、主に以下の 2 つの支援を行う。

支援1. メタデータ作成ツールの DB スキーマの情報を使ってメタデータ記述規則(DSP)のひな形を生成する

支援2. メタデータ記述規則（DSP）の情報を使ってメタデータ作成ツールのひな形を生成する

図 15 は、本研究で提案するメタデータ記述規則設計手法における支援 1 と支援 2 の役割を図に表したものである。支援 1 は Step2 に対する支援を行い、支援 2 は Step1 に対する支援を行う。以下、5.1 節では支援 1 について、5.2 節では支援 2 について詳しく説明する。

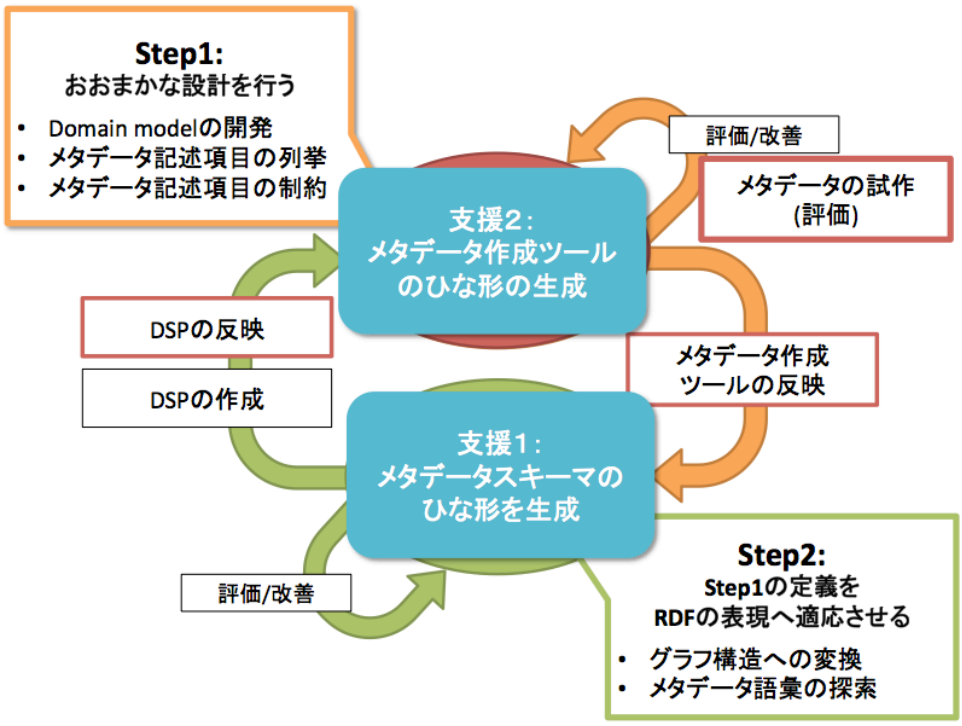


図 15　メタデータ記述規則の設計手法と支援 1,2 の関係図

5.1. 支援 1: メタデータ作成ツールの RDB の情報を用いたメタデータ記述規則のひな形生成

支援 1 では、メタデータ記述規則の設計手法において、メタデータスキーマ設計者が Step1 で開発したメタデータ作成ツールの DB スキーマの情報を利用してメタデータ記述規則のひな形を生成する支援を行う。これにより、メタデータスキーマ設計者は Step2 の作業として必要なグラフ構造の構築とメタデータ語彙の探索を、支援システムを使って容易に行える様になる。5.1.1 項では、DB スキーマを基に RDF のグラフ構造を構築するパターンとそのアルゴリズムについて述べる。5.1.2 項では、RDB が持つテーブル名とカラム名を用いてメタデータ語彙を探索する手法について述べる。

5.1.1. DB スキーマを基にしたグラフ構造の構築パターンとアルゴリズム

本項では、筆者のこれまでのメタデータ記述規則の設計経験を踏まえて、DB スキーマを基にどの様に RDF のグラフ構造を構築するかについて、構築パターンの例を挙げて説明する。また、その構築アルゴリズムについて述べる。なお、グラフとテーブルの関連図の書き方は図 16 の通りであり、以降で登場する N は全て N>1 の自然数を表す。

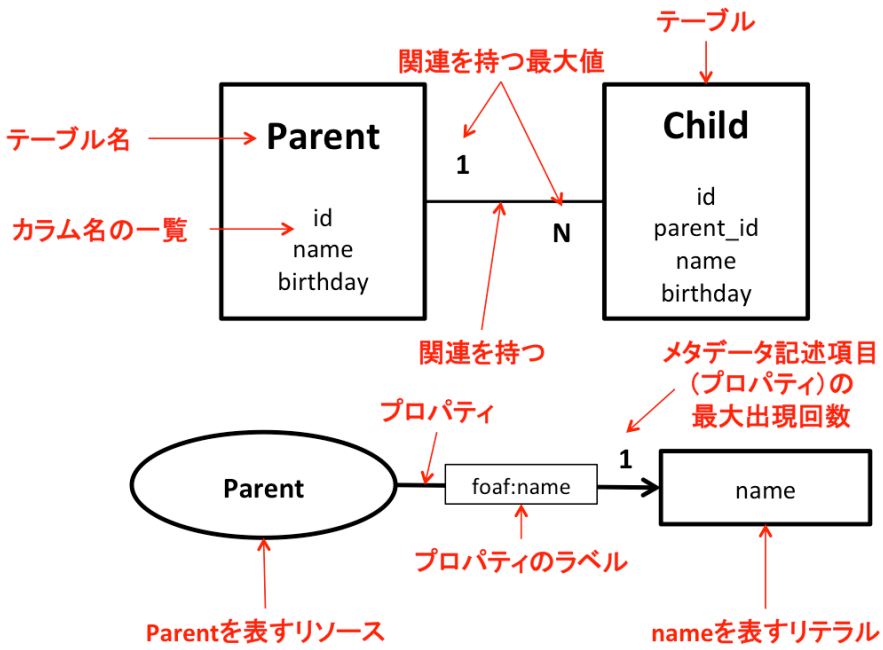


図 16　テーブルとグラフの関連図の書き方

まずテーブル自身がどのようにRDFのグラフ構造へと置き換えられるかについて説明する。Tim Berners-Lee は Relational Detabase and the Semantic Web [25]において、RDB は以下の様に RDF に置き換えられると述べている。

- 1 つのレコードは 1 つの RDF ノードに対応する
- カラム名は RDF のプロパティのタイプに対応する
- セルに入力される 1 つ 1 つの値は、目的語として記述する値に対応する

本研究も、この考えに沿ってグラフ構造の構築を行う。図 17 は関連を持たないテーブルからグラフ構造を構築した例である。これは、以降の構築パターンの基本形となるものである。テーブルはリソースとして、キーを除いたカラムはそのリソースから伸びるプロパティとして置き換える。プロパティの目的語はリテラルであり、テーブルの項目には最大 1 つまでしか値を入力出来ないことから、最大出現回数は 1 とする。図 17 の例では、テーブル Person は関連を持たず、キー以外のカラムに name と birthday を持つため、リソース Person から name と birthday を記述するためのプロパティが 2 つ伸びている。

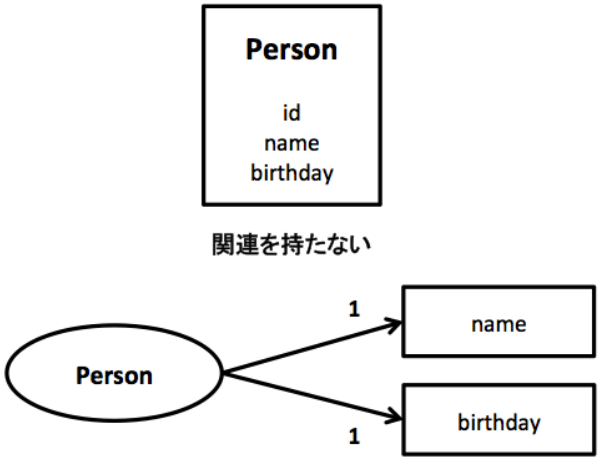


図 17　関連を持たないテーブルの場合のグラフ構造の構築

一方、DB スキーマのテーブル同士の関連には、以下の 3 種類がある。

- 1 対 1
- 1 対多
- 多対多

それぞれの関連をそれぞれ持つ場合に、DB スキーマからどの様にグラフ構造を構築するべきかを考えた。

I. 1 対 1 の場合

2 つのテーブルが 1 対 1 関連を持つ場合は、図 18 の様な 3 つの構築パターンが考えられる。これら 3 つの構築パターンは、関連付けられているテーブル（図 18 ではテーブル Place）をどの様に置き換えるかで大きく異なる。Pattern1 は関連付けられているテーブルを、URI を持つリソースとして置き換えている。Pattern2 は空白ノードという一意の URI を持たないリソースとして置き換えている。Pattern3 は関連付けられているテーブルをリソースとして残さず、そのカラムを直接リソース Organization のプロパティに置き換えている。これらの構築パターンのどれが最も良いかは一概には言えないため、本システムではこれらのパターンの提示のみを行い、状況に基づいてメタデータスキーマ設計者に判断してもらう。構築パターンを選ぶ目安としては、Place を一意に識別したい場合は Pattern1 を、Place を一意に識別する必要はないが、カラム latitude と longitude はセットで扱いたい場合は Pattern2 を、出来るだけ単純なグラフ構造にしたい場合は Pattern3 が望ましいと考えられる。また、関連付けられているテーブルのキーを除いたカラムの数が 1 つの場合は、空白ノードを用いると冗長なグラフ構造となるため、Pattern1 か Pattern3 のどちらかをメタデータスキーマ設計者に選択してもらう。

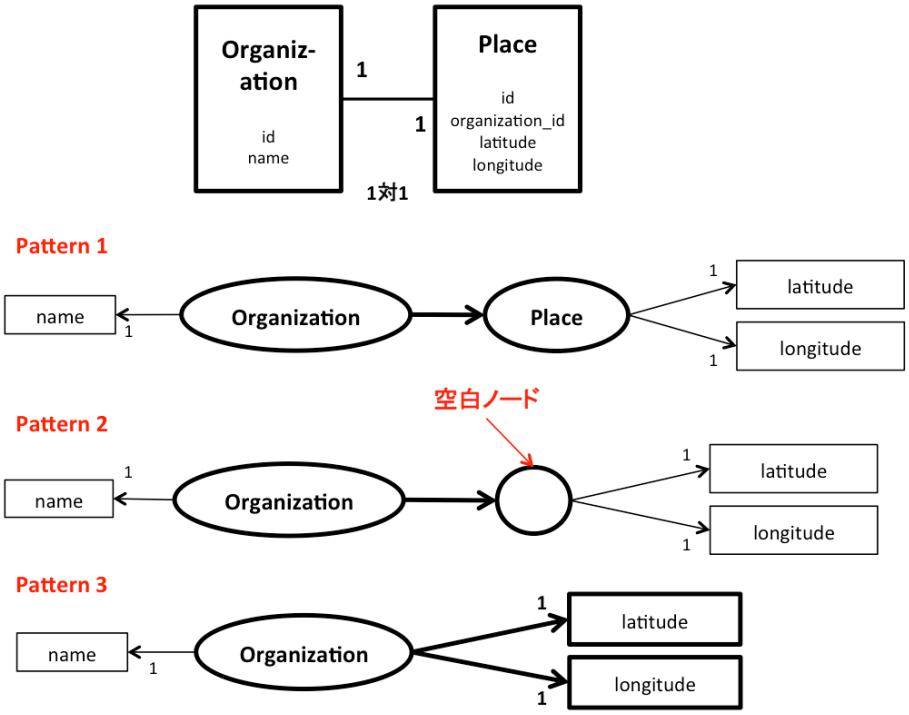


図 18　1 対 1 の場合のグラフ構造の構築

II. 1 対多の場合

2 つのテーブルが 1 対多の関連を持つ場合は、1 対 1 の場合と同じ 3 つの構築パターンが考えられる。この構築パターンを分ける要素は関連付けられているテーブル（図 19、図 20 ではテーブル Member）のキーを除いたカラムの数である。図 19 は、関連が 1 対多で、キー以外のカラムの数が 2 つ以上の場合の構築パターンの例である。この場合、関連付けられているテーブルを空白ノードに置き換える Pattern1 と URI を持つリソースとして置き換える Pattern2 の 2 つの構築パターンが考えられる。また、リソース Organization から空白ノードまたはリソース Member へ伸びるプロパティは最大出現回数が N となる。

図 20 は、関連が 1 対多で、キー以外のカラムの数が 1 の場合の構築パターンの例である。この場合、1 対 1 のときと同様に関連付けられているテーブルを空白ノードに置き換えるパターンは採用しない。メタデータスキーマ設計者は関連付けられているテーブルを、URI を持つリソースとして置き換える Pattern1、または関連付けられているテーブルをリソースとして残さず、そのカラムを直接リソース Organization のプロパティに置き換えた Pattern2 のどちらかを選択する。Pattern2 の場合、テーブル Member のカラム name を置き換えたプロパティの最大出現回数は N となる。

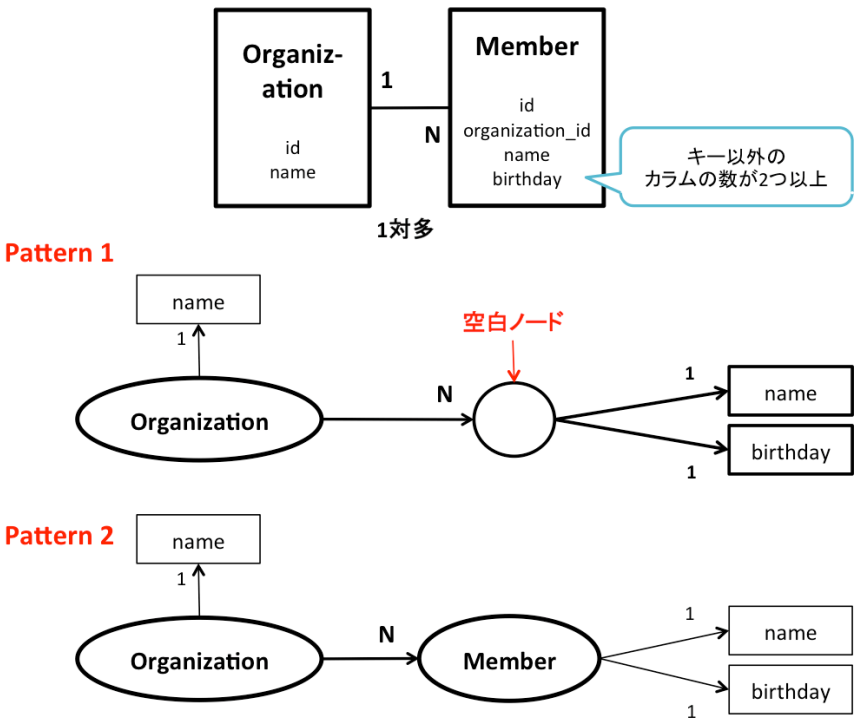


図 19　1 対多でカラムの数が 2 つ以上の場合のグラフ構造への構築

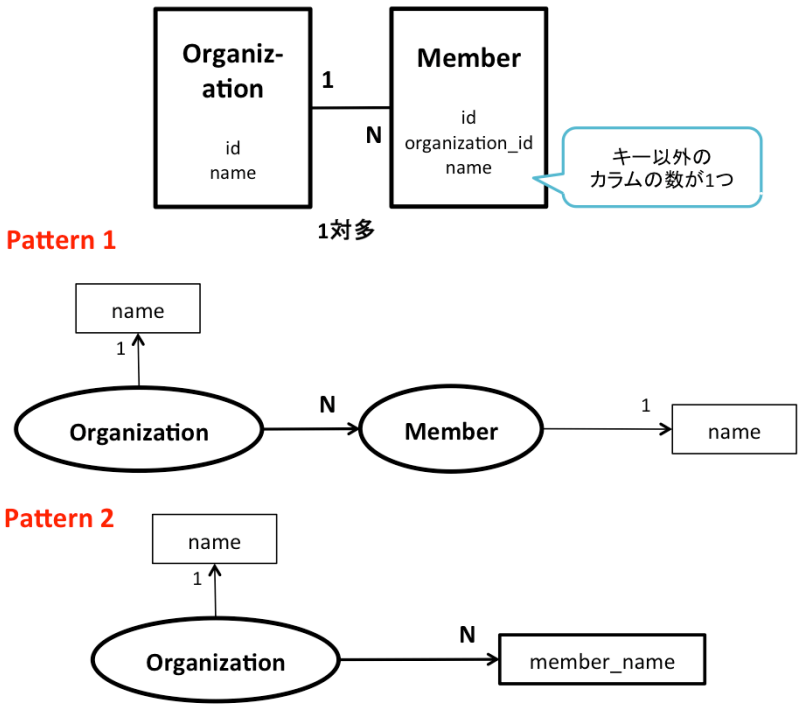


図 20　1 対多でカラムの数が 1 つの場合のグラフ構造への構築

III. 多対多の場合

2 つのテーブルが中間テーブルを使って多対多の関連となっている場合は、図 21 の様なグラフ構造の構築が考えられる。ただし、このような構築パターンになるのは中間テーブル（図 21 の場合はテーブル Book-Author）がキー以外のカラムを持たない場合である。キー以外のカラムを持つ場合は 1 体多の場合と同じ構築パターンとなる。図 21 では、テーブル Book-Author がキー以外のカラムを持たないため、テーブル Book とテーブル Author はそれぞれ、URI を持つリソース Book、リソース Author に置き換えられる。テーブル Book-Author はリソースとしても、カラムとしても残らず、リソース Book とリソース Author は互いに最大出現回数が N となるプロパティを延ばす。

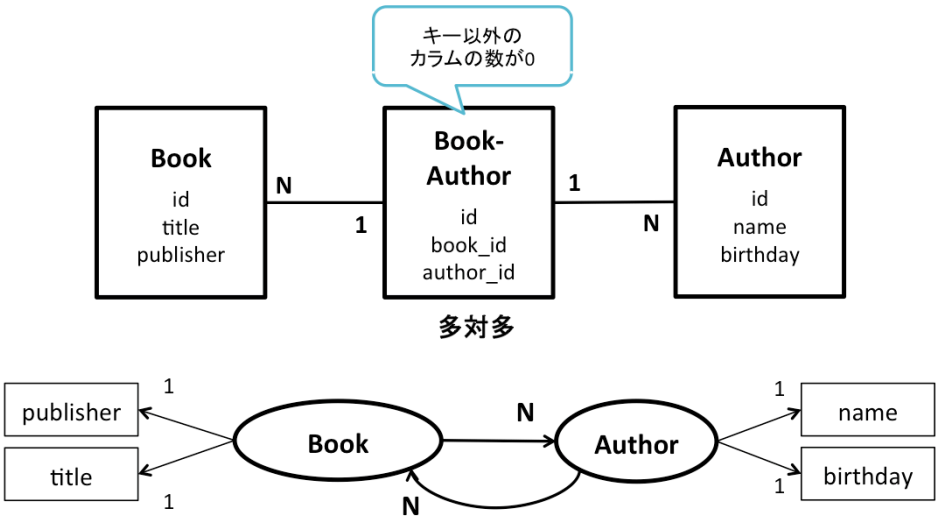


図 21　多対多の場合のグラフ構造への構築

次にこれらの構築パターンの判断を実際に行うためのアルゴリズムについて説明する。また本研究では、Web アプリケーションフレームワークである Ruby on Rails で開発されたアプリケーションの RDB の情報を読み込むことを前提としている。(Ruby on Rails については詳しくは 6 章で述べる。) Ruby on Rails では Active Record[9]というフレームワークを用いて、テーブルをモデルクラス（以下、モデルと呼ぶ）と見なす。また、テーブル間の関連（アソシエーション）を表 2 のいずれかを使って表現出来る。そのため本アルゴリズムでは、モデルが表 2 のいずれかの関連を持つということを事前に分かっていることを前提とする。

表 2　Active Record の関連一覧

関連の種類	説明
belongs_to	一方のモデルに 1 対 1 で従属する関連
has_one	1 対 1 の関連で一方のモデルを持つ（格納する）関連
has_many	1 対多の関連で一方のモデルを持つ（格納する）関連
has_many :through	中間モデルを介した多対多の関連
has_one :through	中間モデルを介した 1 対 1 の関連
has_and_belongs_to_many	中間モデルを介さない多対多の関連 （※中間テーブルは存在する）

[9] http://guides.rubyonrails.org/association_basics.html

本アルゴリズムでは、構築パターンの説明で述べたリソースやプロパティの代わりに、それらの制約を記述するための Description Template と Statement Template を構築していく。また、本アルゴリズムを動かす前提として、それぞれのモデルが互いにどの様な関連を持つかは全て取得済みであるとする。図 22 はこのアルゴリズムのメインとなる部分である。ここでは、まず Description Template の集合となる Description Set Template（DST）を作成する。次に全てのモデル（Model）をチェックし、他のモデルに従属の関連を持っていない場合は、そのモデルのための Description Template（DT）を作成する。また、このアルゴリズムは再帰的に関連を辿って行く。そこで、無限ループを防ぐために、developedDescriptionTemplates という配列に作成した Description Template を格納しておく。developeStatementTemplates()では、Description Template が持つ Statement Template の情報を作成していく。全ての情報を読み込んだ後、Description Set Template に Description Template を格納する。

```
#MAIN

DST = DescriptionSetTemplate.New
Check each Model of all Models
   NEXT roop If Model is belongs to other Model
   DT = DescriptionTemplate.New(Model)
   Push DT into developedDescriptionTemplates
   developStatementTemplates(DT,Model)
   Push DT into DST.description_templates
end
```

図 22　RDB スキーマからの RDF のグラフ構造の構築：MAIN アルゴリズム

図 23 は developStatementTemplates()のアルゴリズムを示したものである。ここでは、カラムの名前を確認し、キー以外の場合は Statement Template を作成する。（このとき、Ruby on Rails のアプリケーションでは、作成日、更新日というカラムが自動的に追加されてしまうため、正確にはそれらのカラムも除いている。）また、RDB のカラムは 1 度しか記述することが出来ないため、最大出現回数は 1 である。そして、Description Template にそれらを格納していく。カラムの確認が終わったら、他のモデルとの関連を確認する。本アルゴリズムでは、モデルが、どの関連でどのモデルと繋がっているかを事前に取得している。それぞれの関連毎に繋がるモデルが存在する場合は、transformModelToDescriptionTemplate()を実行し、それ以外の場合は何もしない。また全ての行程が終わった後に、余計な空の配列がある場合は削除しておく。

```
def developStatementTemplates(DT, Model)
  Check each Column of Columns of Model
    If Column.name is NOT primary key or foreign key
        ST = StatementTemplate.New(Column)
        ST.max_qualified_cardinality = 1
        Push ST into DT.statement_templates
    end
  end

  If Model has “has_one” relation then
    transformModelToDescriptionTemplate (DT,Model,”has_one”)
  end
  If Model has “has_many” relation then
    transformModelToDescriptionTemplate (DT,Model,”has_many”)
  end
  If Model has “has_one_belongs_to_many” relation then
    transformModelToDescriptionTemplate (DT,Model,”
has_and_belongs_to_many”)
  end
  Remove null Element from DT.statement_templates
end
```

図 23　developStatementTemplates() アルゴリズム

図 24 の transformModelToDescriptionTemplate()では、モデル同士の関係を表すための Statement Template を作成する。このアルゴリズムでは、主に表 2 に示した「has_many :through」や「has_one :through」、「has_and_belongs_to_many」、自己参照などのいずれの関係を持つかを判断している。関連モデル（RelatedModel）とモデル（Model）の名前が一致すれば自己参照であり、Statement Template（ST）の目的語に自分自身のモデルについての Description Template を結ぶ。一致しない場合は、まず関連モデルについての Description Template が既に作成済みでないかを確認する。作成済みの場合、作成された Description Template を Statement Template の目的語として結ぶ。更に関連が「has_many :through」、「has_one :through」であり、仲介役となる中間モデル（JunctionModel）が存在し、そのモデルのキー以外のカラムが 0 の場合、関連モデルについての Description Template と結びつける。中間モデルのキー以外のカラムが 1 以上の場合は、中間モデルについての Description Template と結ぶ。このとき、関連モデル自身が中間モデルではないかを確認し、そうでない場合は更に構築パターンを細かく判断するために jugePattern()を行う。関連モデルが中間モデルであり、キー以外のカラムの数が 1 以上の場合は、関連モデルの Description Template を Statement Template の目的語として結び付ける。0 の場合は Statement Template は作成されない。

図 25 の storeDescriptionTemplate()では、まず関連モデル（RelatedModel）の

Description Template（ST.related_dt）がリソースに URI を持つかどうかを判断する。そして、無限ループを避けるため、2 回以上同じ動作を行わないために、作成した関連モデルの Description Template を、作成済みの Description Template を記録するための配列 developedDescriptionTemplates に格納する。更に再帰的に developedDescriptionTemplates()を実行し、関連モデルの Description Template の Statement Template を作成していく。それらが終了すると、関連モデルと元のモデルを結ぶ Statement Template（ST）を、元のモデルの Description Template（DT）に格納する。

```
def transformModelToDescriptionTemplate (DT,Model,Association)
  Check each RelatedModel
    ST = StatementTemplate.New(RelatedModel)
    Unless RelatedModel.name = Model.name then
      Unless RelatedModel is already developed as DescriptionTempalte then
        ST.related_dt = DescriptionTemplate.New(RelatedModel)
        Unless RelatedModel has :through Association and there is JunctionModel
then
          Unless RelatedModel has a role of JunctionModel
            judgePattern(DT,ST,RelatedModel,Association)
          else
           If RelatedModel.column_number >= 1 then
            storeDescriptionTemplate(DT,ST,true,RelatedModel)
           end
          end
        else
          If JunctionModel.column_number < 1 then
            storeDescriptionTemplate(DT,ST,true,RelatedModel)
          end
        end
      else
        ST.related_dt = debelopedDescriptionTemplate(RelatedModel)
        Push ST into DT.statement_templates
      end
    else
      ST.related_dt = DT
      Push ST into DT.statement_templates
    end
  end
end
```

図 24　transformModelToDescriptionTemplate()アルゴリズム

```
def storeDescriptionTemplate(DT,ST,id_presence,RelatedModel)
   If id_presence = true then
      ST.related_dt has URI
   end
   Push ST.related_dt into developedDescriptionTemplates
   developStatementTemplates(ST.related_dt, RelatedModel)
   Push ST into DT.statement_templates
end
```

図 25　storeDescriptionTemplate()アルゴリズム

図 26 の judgePattern()では、表 2 に示した「has_many」、「has_one」、「has_and_belongs_to_many」のいずれかの関連で結ばれた関連モデル（RelatedModel）の構築パターンを判断する。まず、askIDPresence()でメタデータスキーマ設計者に対して、関連モデルをリソースに置き換えた場合に URI を持たせるかどうかを問い合わせる。このとき、関連（Association）が「has_and_belongs_to_many」の場合は、必ず URI を持たせるようにする。次に関連モデルのキー以外のカラムの数を調べる。キー以外のカラムの数が 1 の場合、URI を持たせる場合は Statement Template（ST）の目的語に関連モデルの Description Template を結びつけ、URI を持たせない場合は、他に関連モデルが別のモデルと関連を持たないことを確認した上で、transformModelToStatementTemplate()で関連モデルが持つカラムを元のモデルの Statement Template として結びつける。別のモデルと関連を持つ場合は、Statement Template（ST）の目的語に関連モデルの Description Template を結びつける。カラムの数が 0 の場合でも、更に他のモデルと関連している可能性があるため、Statement Template の目的語に関連モデルの Description Template を結ぶ。カラムの数が 2 以上の場合も同様に、URI を持てば Statement Template の目的語に関連モデルの Description Template を結びつける。URI を持たない場合は、他に関連モデルが更に別のモデルと関連を持たないことをまず確認する。関連を持つ場合は、Statement Template（ST）の目的語に関連モデルの Description Template を結びつける。持たない場合は、元のモデルと関連モデルの関連が「has_one」かそうでないかを確認する。「has_one」の場合は、更に構造を持たせるかどうかをメタデータスキーマ設計者に問い合わせる。このとき、持たせるならば関連モデルのリソースは空白ノードとして記述する。構造も持たせない場合は、transformModelToStatementTemplate()で関連モデルが持つカラムを全て元のモデルの Statement Template とする。関連が「has_one」でなかった場合は、関連モデルのリソースは空白ノードとなり、元のモデルの Statement Template の目的語には関連モデルの Description Template が結ばれる。

図 27 の transformModelToStatementTemplate()では、developeStatementTemplates()

と同様に関連モデル（RelatedModel）のカラムを確認し、Statement Template を作成している。developeStatementTemplates()との大きな違いとしては、カラムを全て関連モデルの Description Template ではなく、元のモデルの Description Template（DT）に格納している点で異なっている。

本システムでは、これらのアルゴリズムを用いてグラフ構造の構築の支援を行う。5.1.2 項では、更にメタデータ語彙の探索支援について説明する。

```
def judgePattern(DT,ST,RelatedModel,Association)
  id_presence = true
  askIDPresence(RelatedModel)  Unless Association = “has_and_belongs_to_many”
  If RelatedModel,column_number > 1 then
    If id_presence = true then
      storeDescriptionTemplate(DT,ST,id_presence,RelatedModel)
    else
      If RelatedModel has other Associations with other Models then
        storeDescriptionTemplate(DT,ST,id_presence,RelatedModel)
      else
        If Association = “has_one” then
          structure_presence = askStructurePresence(RelatedModel)
          If structure_presence = true then
            storeDescriptionTemplate(DT,ST,id_presence,RelatedModel)
          else
           transformModelToStatementTemplate(ST.related_dt,DT,RelatedModel,
Association)
          end
        eles
          storeDescriptionTemplate(DT,ST,id_presence,RelatedModel)
        end
      end
    end
  else If RelatedModel,column_number = 1 then
    If id_presence = true then
      storeDescriptionTemplate(DT,ST,id_presence,RelatedModel)
    else
      If RelatedModel has other Associations with other Models then
        storeDescriptionTemplate(DT,ST,id_presence,RelatedModel)
      else
        transformModelToStatementTemplate(DT,RelatedModel, Association)
      end
    end
  else
    storeDescriptionTemplate(DT,ST,id_presence,RelatedModel)
  end
end
```

図 26　judgePattern()アルゴリズム

```
def transformModelToStatementTemplate(DT,RelatedModel,Association)
  Check each Column of Columns of RelatedModel
    If Column.name is NOT primary key or foreign key
        ST = StatementTemplate.New(Column)
        ST.max_qualified_cardinality = 1
        Push ST into DT.statement_templates
    end
  end
end
```

図 27　transformModelToStatementTemplate()アルゴリズム

5.1.2. メタデータ語彙の探索手法

本項では RDF で表現するために必要なメタデータ語彙を、DB スキーマの情報を利用してどの様に探索するかについて説明する。

5.1.1 項で述べた様に本研究では、DB スキーマから RDF のグラフ構造を構築する場合、テーブルそのものをリソース、テーブルのキー以外のカラムをリソースから伸びるプロパティとして置き換えている。そのため、各テーブルに付けられている名前は一般的にそのテーブルには何の情報が記述されているのかを表す情報であると考えられる。そして、カラム名はそのテーブル名が示すものに対する各要素を表す情報である。例えば図 28 の場合、テーブル Person は「Person（人）」についての情報を記述するテーブルであり、そのカラム name、birthday はそれぞれ「Person の name（人の名前）」、「Person の birthday（人の生年月日）」をそれぞれ記述する。そこで本研究ではテーブル名を、リソースのクラスを探索するためのキーワードとして、カラム名を、リソースから伸びるプロパティを探索するためのキーワードとして用いることにした。またリソース同士を繋ぐプロパティは、対応するカラム名が存在しないため、リソースとして関連付けられるテーブル名を探索キーワードとして利用する。

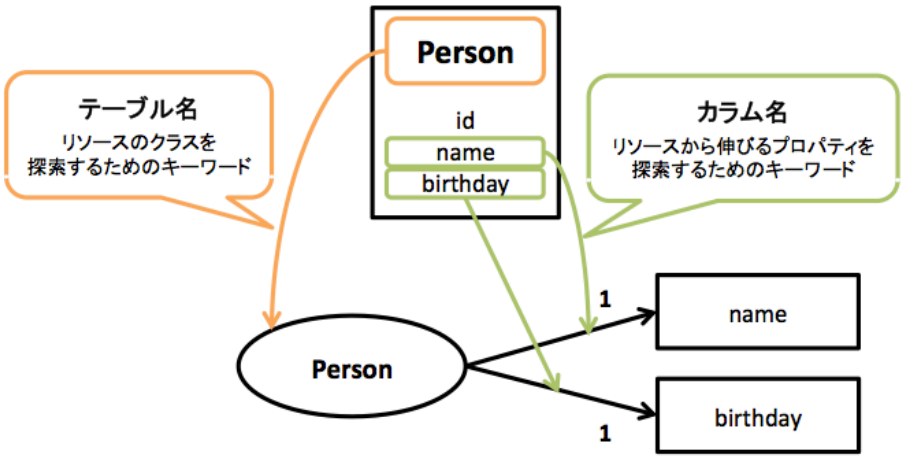


図 28　メタデータ語彙の探索に必要なキーワードの割当

また現在、メタデータ語彙を検索するためのサービスとして Linked Open Vocabularies（LOV）[10]がある。LOV はメタデータ語彙定義を収集・蓄積し、検索キーワードを入力することで、そのキーワードを含むクラスやプロパティの一覧を検索結果としてユーザに返す。また、メタデータ語彙定義を検索するための SPARQL Endpoint を 2 つ公開している。1 つはメタデータ語彙定義に対するメタデータのデータセット（Linked Open Vocabularies Endpoint, LOV-E）[11]、もう一つはメタデータ語彙定義を集めたデータセット（LOV Aggregator Endpoint, LOVA-E）[12]である。本研究では、これらのデータセットを利用してメタデータ語彙の探索を行う。

LOV-E は 339 件（2014 年 1 月 7 日現在）のメタデータ語彙定義についてのメタデータを持つ。そのメタデータ語彙定義の提供者や更新日、プロパティとクラスの数などを含んでいる。また LOV では、これらのメタデータ語彙定義をカテゴリに分けており、メタデータ語彙定義がどのカテゴリに含まれるかという情報も含んでいる。支援システムでは、メタデータ語彙を探索する際に、このカテゴリを用いてメタデータ語彙の検索結果を絞ることが出来るようにした。LOV のカテゴリは表 3 の通りである。

LOVA-E は、339 件（2014 年 1 月 7 日現在）のメタデータ語彙定義を集めたデータセットである。ここでは、2.2.1 項で説明したようなメタデータ語彙についての基本的な定義と制約についてのメタデータが提供されている。加えて、そのメタデータ語彙が実際に LOD 上で、どの様に利用されているかを調べた統計情報も取得することが出来る。本研究では

[10] http://lov.okfn.org/dataset/lov/
[11] http://lov.okfn.org/endpoint/lov
[12] http://lov.okfn.org/endpoint/lov_aggregator

その中でも、あるメタデータ語彙が LOD 上で利用されている数（LOD popularity）を、探索したメタデータ語彙のランキングに用いた。

図 29 と図 30 は、それぞれプロパティ、クラスを探索するために用いた SPARQL 文である。これらは、Honma らの研究[27]を参考に作成した。赤字で書かれた category_url の部分には、ユーザが選択したカテゴリの URI が入る。同様に keyword の部分には図 29 ではカラム名、図 30 ではテーブル名が含まれる。これらの SPARQL 文は、どちらもカテゴリで検索対象のメタデータ語彙定義を絞り込んだ上で、プロパティ、クラスと識別されるリソースから伸びるプロパティの目的語に、キーワードが含まれているものを LOD popularity が大きい順に返す。なお LOD popularity が 0 のものは検索結果に含まない。

本支援システムでは、5.1.1 項と本項で説明した方法で、メタデータ記述規則のひな形を生成する。これを実際に実装したものに関しては 6 章で詳しく述べる。その前に次節では、支援 2 のメタデータ作成ツールのひな形を生成するための方法について説明する。

表 3　LOV のメタデータ語彙カテゴリ

大カテゴリ	大カテゴリの概要	小カテゴリ	小カテゴリの概要
Metadata	Vocabularies used for metadata and annotations, such as Dublin Core	Quality	Quality, Provenance and Trust
		Tagging	Vocabularies describing tags, folksonomies, tagging events
Science	Science Facts and Figures	Life Sciences	Biology and Life Sciences
		Sustainability and environmental management	Energy, Natural resources, Biodiversity, ...
		Methods and Protocols, Units, Measures	Vocabularies used to describe science methods and protocols, numerical data, statistics, measures etc.
		Health	Vocabularies for healthcare and medicine
Data, Resources and Protocols	Quality, identitification, description, protocols, API	RDF Data	RDF data bases, named graphs, RDB to RDF ...
		Protocols	Protocols, Services, API ...
		PLM	Product Lifecycle Management
		Social Semantic Desktop	Ontologies developed by the NEPOMUK Project
		Security	Security, Network, attacks and countermeasures
General and Upper	General vocabularies and top ontologies	General and Upper Ontologies	Ontologies for general use in many domains
		Support vocabularies	Miscellaneous general support vocabularies
		PROTON Ontologies	Ontologies developed in the PROTON project
		Schema.org	Schema.org and extensions
Media	Press, Sound, Music, Image, Video	Image and Video	Image and video description
		Music and Sound	Music, Sound, Audio files
		Multimedia	Video, TV, Broadcasting ...
		Press and News	Press and News Vocabularies
Space, TIme, Events	Geography, Calendar, History, Events	Geography	Geographical entities and features
		Geometry	Geometry and spatial relationships

		Events	Vocabularies for all kind of events : political, cultural, sports ...
		Timeline, Calendar	Time intervals, timeline, time zones
The City	People and Society	The Academy	Vocabularies for academics and research
		Social organization	Social relations, collectivities, organizations
		The Government	Governement, Administrative organisation
		People	Personal stuff : vcards, carreer, genealogy, interests
The Library	Libraries, catalogues, knowledge organization systems	Vocabularies, Languages and Terminologies	Vocabularies used to describe and organize vocabularies, languages and terminologies
		Archives, Catalogs, Classification systems	Vocabularies used to describe and organize library and archives resources
		FRBR, RDA, FRAD and related vocabularies	Vocabularies built on the FRBR model
		Semantic Publishing and Referencing Ontologies	The Semantic Publishing and Referencing Ontologies (SPAR) form a suite of orthogonal and complementary ontology modules for creating comprehensive machine-readable RDF metadata for all aspects of semantic publishing and referencing. A sourceforge repository for all vocabularies in this suite is at http://sempublishing.svn.sourceforge.net/viewvc/sempublishing/
The Market Place	Products and services	Industry	Industrial products and process
		E-Business	Vocabularies supporting online business
		Travel	Transport and Tourism
		Contracts	Calls, contracts, payments
		Recommandations	Review and Recommendations
		Food	Vocabularies for food description

```
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX dcterms:<http://purl.org/dc/terms/>
PREFIX voaf:<http://purl.org/vocommons/voaf#>
SELECT distinct ?term ?nameUri ?prefix ?lod_popularity
WHERE {
 { ?term rdfs:subPropertyOf* rdf:Property . }
 UNION
 {
   ?c rdfs:subPropertyOf* rdf:Property .
   ?term rdf:type ?c .
 }
 ?term rdfs:isDefinedBy ?nameUri .
 ?term ?p ?o.
 ?term voaf:occurrencesInDatasets ?lod_popularity.
   SERVICE <http://lov.okfn.org/endpoint/lov>
   { SELECT *
     WHERE{
     <category_url> dcterms:hasPart* ?nameUri.
     ?nameUri <http://purl.org/vocab/vann/preferredNamespacePrefix> ?prefix.
     }
   }
  FILTER isIRI(?term) .
  FILTER REGEX(str(?o), ¥".*keyword.*¥", ¥"i¥") .
}
ORDER BY DESC(?lod_popularity)
```

図 29　プロパティを探索するための SPARQL 文

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dcterms:<http://purl.org/dc/terms/>
PREFIX voaf:<http://purl.org/vocommons/voaf#>
SELECT distinct ?term ?nameUri ?prefix ?lod_popularity
WHERE {
  { ?term rdfs:subClassOf* rdfs:Class . }
  UNION
  {
    ?c rdfs:subClassOf* rdfs:Class .
    ?term rdf:type ?c .
  }
  ?term rdfs:isDefinedBy ?nameUri .
  ?term ?p ?o.
  ?term voaf:occurrencesInDatasets ?lod_popularity.
  SERVICE <http://lov.okfn.org/endpoint/lov>
  { SELECT *
    WHERE{
    <category_url> dcterms:hasPart* ?nameUri.
    ?nameUri <http://purl.org/vocab/vann/preferredNamespacePrefix> ?prefix.
    }
  }
  FILTER isIRI(?term) .
  FILTER REGEX(str(?o), ¥".*keyword.*¥", ¥"i¥") .
}
ORDER BY DESC(?lod_popularity)
```

図 30　クラスを探索するための SPARQL 文

5.2. 支援２：メタデータ記述規則を基にしたメタデータ作成ツールのひな形生成

支援 2 ではメタデータ記述規則の設計手法において、メタデータの試作のために用いるメタデータ作成ツールのひな形を、メタデータ記述規則を基に生成する支援を行う。具体的には、DSP に記述されたグラフ構造から DB スキーマを構築し、RDB を用いたメタデータ作成ツールのひな形を生成出来るようにする。これにより、メタデータスキーマ設計者は、容易にメタデータ作成ツールを開発し、メタデータの試作を行うことが出来る様になる。なお、メタデータ作成ツールのひな形は、メタデータ作成ツールが持つ DB スキーマの情報や、RDB に保存されたデータを RDF として出力するための情報を含む Ruby のコードとして出力される。これについては 6 章で詳しく説明する。本節では、特にメタデータ記述規則が持つグラフ構造の情報（構造的制約）から、どの様に DB スキーマを構築するかに重点を置き、その構築パターンとアルゴリズムについて説明する。

まず、これから述べていく構築パターンの基本である、リソースをテーブルに置き換える方法について述べる。リソースの場合、プロパティ（メタデータ記述項目）の最大出現回数が構築パターンを分ける重要な要素であり、最大出現回数が 1 回であるか 2 回以上であるかで構築パターンが変わる。リソースから伸びる全てのプロパティの最大出現回数がすべて 1 回のとき、グラフ構造は 5.1.1 項の図 17 とは逆に DB スキーマへ置き換えられる。リソースはテーブルとなり、プロパティはそのテーブルのカラムとなる。プロパティの最大出現回数が 2 回以上のものを含む場合、そのプロパティは図 31 の様に 1 対多でテーブルとして関連付けられる。

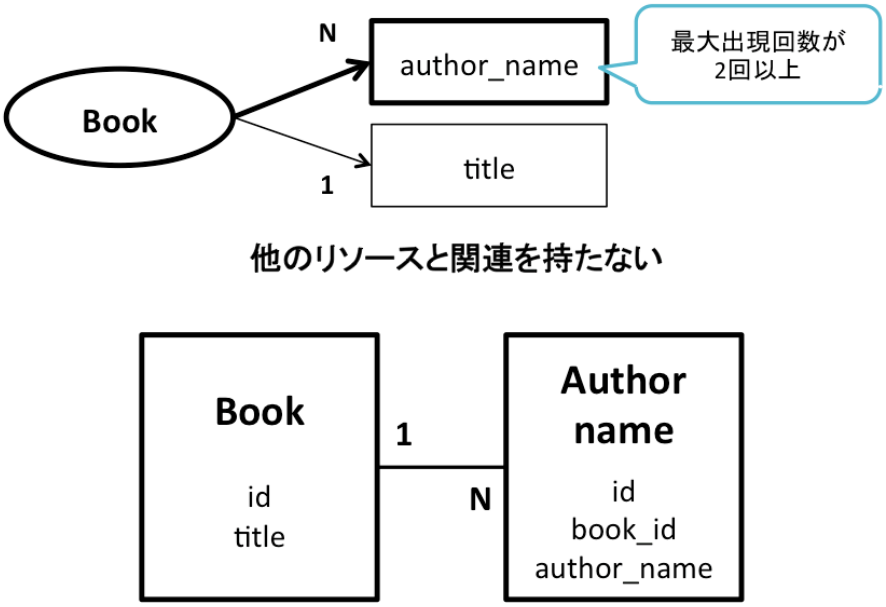


図 31　関連を持たず最大出現回数が 2 回以上のプロパティを含む場合の構築

次に、メタデータ記述規則では、DB スキーマと同様にリソース同士が以下の様な関連で結ばれていると考えられる。

- 1 対 1
- 1 対多
- 多対多

次に、これらの関連を持つ RDF のグラフ構造からどの様に DB スキーマが構築されるべきかを考えた。

I.　1 対 1 の関連を持つ場合

2 つのリソースが 1 対 1 で関連を持つ場合、関連付けられているリソースが URI を持つか否か、即ち空白ノードかどうかによって構築パターンが異なる。図 32 の様に、空白ノードと 1 対 1 で関連を持つ場合、Pattern1 の様に空白ノードをテーブルとして置き換え 1 対 1 で関連付ける場合と、Pattern2 の様に空白ノードから伸びるプロパティをカラムとして置き換える場合が考えられる。

図 33 の様に、URI を持つリソースと 1 対 1 で関連を持つ場合は図 32 の Pattern1 と同様に、関連付けられているリソースをテーブルとして置き換え 1 対 1 で結びつける。

II. 1 対多の関連を持つ場合

2 つのリソースが 1 対多で関連を持つ場合、図 34 の様に関連付けられているリソースが URI を持つか否か、即ち空白ノードであるかどうかに関わらず、関連付けられているリソースをテーブルとして置き換え 1 対多の関連で結びつける。

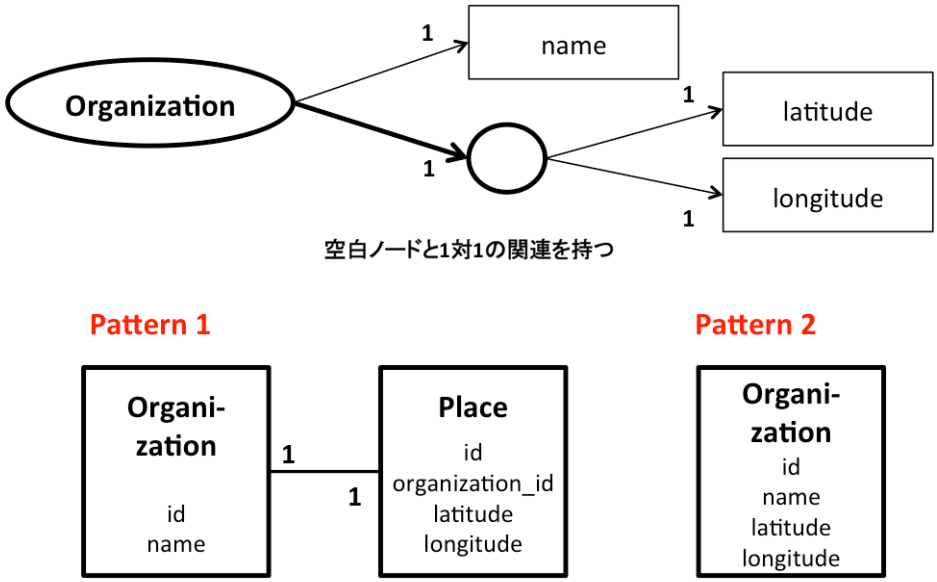


図 32　空白ノードと 1 対 1 の関連を持つ場合の構築

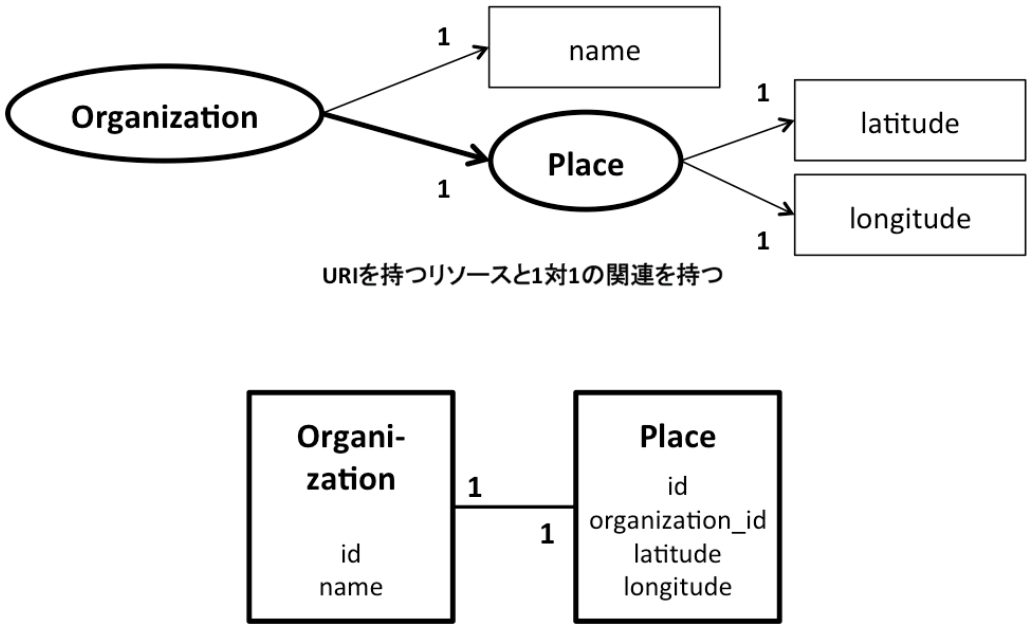


図 33　URI を持つリソースと 1 対 1 の関連を持つ場合の構築

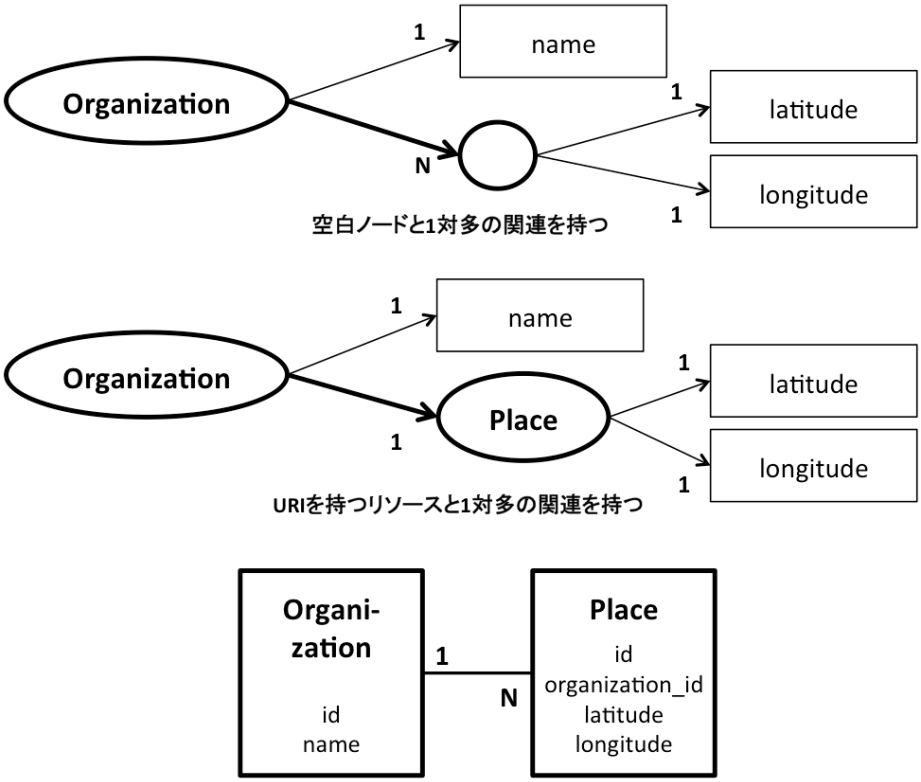


図 34　リソース同士が 1 対多の関連を持つ場合の構築

III. 多対多の関連を持つ場合

URI を持つ 2 つのリソースが多対多で関連を持つ場合は、5.1.1 項の図 21 と逆の構築パターンとなり、それぞれのリソースをテーブルに置き換え、更に中間テーブルを追加することで多対多の関連を作る。また本研究では、空白ノードが多対多で関連を持つ場合に関しては想定していない。

次にこれらを実際に行い、メタデータ作成ツールのための DB スキーマを構築するためのアルゴリズムについて説明する。本アルゴリズムでは 5.1.1 項で説明したアルゴリズムと同様に Ruby on Rails 上で ActiveRecord を用いることを前提とし、テーブルの代わりにモデルを生成する。なお、実際のプログラムでは、正確にはモデルを生成するのではなく、そのための Ruby スクリプトを出力している。また、本アルゴリズムを動かす前提として、Description Template が互いにどの様な関連を持つかは取得済みの状態となっている。特に多対多の関連を持つ Description Template は、その組み合わせを ManyToManyCombinations という配列に事前に格納している。但し、この配列では対と

なる組み合わせ（例：A と B、B と A）のものがそれぞれ含まれている。

図 35 はこのアルゴリズムのメインとなる部分である。ここでは、まず先ほど述べた ManyToManyCombinations に格納された Description Template の組み合わせに対して、developMany-to-manyRelation()で多対多の関係のモデル（Model）を作成する。その後、他の Description Template に対して属する関係を持たず、グラフの頂点となっている Description Template に対して、developModels()でモデルを作成していく。この時、Description Template に対して属する関連を持っている Description Template のモデルは developModels()を再帰的に行うことで生成していく。

```
# MAIN

Check each ManyToManyCombination of ManyToManyCombinations
  developMany-to-manyRelation(ManyToManyCombination)
end

Check each DT of DescriptionSetTemplate
  Unless DT is belongs to other DT then
    ParentDTs = Array.new
    BrotherDTs = Array.new
    developModel(DT,null,ParentDTs,BrotherDTs)
  end
end
```

図 35　RDF のグラフ構造から RDB スキーマの構築：MAIN アルゴリズム

図 36 は developMany-to-manyRelation()のアルゴリズムを示したものである。ここでは、事前に多対多の関係にある 2 つの Description Template、DT と RelatedDT、また DT が他の Description Template に属するものである場合は、属している Description Templat（ParentDT）を読み込んだ状態から始まる。また下準備として、DT のモデルと RelatedDT のモデルを仲介する中間モデルの名前（JunctionModelName）を、DT の名前と RelatedDT の名前を順に結合して作成する。それと同時に、対となる組み合わせの名前（reversJunctionModelName）も作成しておく。次に developRecordSet()で DT のモデルのカラムを作成する。この時、RelatedDT の外部キーを持っている場合は外部キーを削除しておく。DT のモデルを生成し、reversJunctionModelName と同じ名前のモデルが既に生成されていなければ、JunctionModelName という中間モデルも作成する。relateModelToParentModel で RelatedDT のモデルと表 2 に示した「has_many :through」の関連を、中間モデルと「has_many」の関連を作る。最後に、無限ループを防ぐために既に生成したモデルを配列 developedModels に格納し記録しておく。

```
def developMany-to-manyRelation(ManyToManyConbination)
   DT = one DescriptionTemplate of ManyToManyConbination
   RelatedDT = the other DescriptionTemplate of ManyToManyConbination
   ParentDT = the DesctiprionTemplate which has “One-to-Many” or “One-to-one”
relation with DT
   MediateModelName = DT.name + “_” + RelatedDT.name
   reversMediateModelName = RelatedDT.nam + “_” + DT.name

   RecordSet = developRecordSet(DT,ParentDT)
   If Model(DT) has ForeignKey of RelatedDT then that ForeignKey is deleted

   developNewModel(DT)
   Unless Model, name of which is reversMediateModelName then
      developNewModel(MediateModelName)
   else
      MediateModelName = reversMediateModelName
   end

   relateModelToParentModel(Model(DT),Model(RelatedDT))
   relateModelToParentModel(Model(DT),Model(MediateModelName))

   Push Model(DT) into developedModels
   Push Model(MediateModelName) into developedModels
end
```

図 36　developMany-to-manyRelation()アルゴリズム

図 37 は、developModel()のアルゴリズムを示したものである。ここでは、ある Description Template（DT）と DT が属する関係にある Description Template（ParentDT）（ParentDT が存在しない場合は null）を読み込み、DT のモデルを作成していく。また、DT と ParentDT を実際に関連付けていく。始めに、developRecordSet()で DT のモデルのカラムを生成する。そして、ParentDT が存在するかどうかを確認する。存在しない場合はそのまま DT のモデルを生成する。存在する場合は、DT のモデルが既に作成済みでないかを確認する。作成済みでない場合は、DT のモデルを生成した上で、ParentDT のモデルと 1 体 1 の関係ならば「has_one」、1 対多の関係ならば「has_many」で関連付ける。作成済みの場合は、関連付けだけを行う。developedModels に作成した DT のモデルを記録したら、更に DT から関連している Description Template（RelatedDT）を確認する。RelatedDT を格納している配列 RelatedDTs は、developRecordSet()を行った際に作成したものである。このとき、RelatedDT と DT の名前が同じ場合は、DT に自己参照の関連を作る。それ以外の場合は、RelatedDT に対して developModel()を再帰的に実行していく。

```
def developModel(DT,ParentDT)
  RecordSet = developRecordSet(DT,ParentDT)

  Unless DT has ParentDT then
    Unless Model of DT is already developed then developNewModel(DT)
    relateModelToParentModel(Model(DT),Model(ParentDT))
  else
    developNewModel(DT)
  end

  Push Model(DT) into developedModels
  Check each RelatedDT of RelatedDTs
    Unless RelatedDT.name = DT.name then
      developModel(RelatedDT,DT)
    else
      developSelfReferenceRelation(RelatedDT)
    end
  end
end
```

図 37　developModel()アルゴリズム

図 38 は、developRecordSet()のアルゴリズムを示したものである。ここでは、読み込んだ Description Template（DT）のモデルのカラムを作成する。まず、DT がリソースは URI を持つという制約を持っている場合、URI を入力するためのカラムを作成する。そして、DT の Statement Template（ST）を調べていく。もし、ST が目的語として別の Description Template（RelatedDT）を持つ場合は、ST の最大出現回数から 1 対 1 か 1 対多かを調べる。更に RelatedDT と DT の名前が一致し、自己参照である場合は DT 自身を参照するための外部キーを追加する。そして、DT が関連を持つ DescriptionTemplate を記録する配列 RelatedDTs に RelatedDT を格納していく。RelatedDT が存在しない場合は、最大出現回数を調べ、最大出現回数が 1 のとき DT のモデルのカラムとして ST を追加する。最大出現回数が 2 回以上のときは、更に新たなモデルを生成し、DT のモデルと 1 対多で関連付ける。Statement Template を全て確認したら、次は DT が属する関係にある Description Template（ParentDT）の存在を確認する。ParentDT がある場合は、その外部キーをカラムとして追加する。

本システムでは、これらのアルゴリズムを用いてグラフ構造から DB スキーマの構築を行う。また、構築した DB スキーマを持つメタデータ作成ツールを開発するためのひな形を生成する。6 章では、これらを踏まえて実際に開発した支援システムについて説明していく。

```
def developRecordSet(DT,ParentDT)
  If Resource of DT has URI then Add Column of DT_uri to Model(DT)

  Check each ST of DT.statement_templates
    If ST relates DT to RelatedDT then
      If ST.max_qualified_cardinality = 1 then
        Model(DT) has “One-to-many” relation with Model(RelatedDT)
      else
        Model(DT) has “One-to-one” relation with Model(RelatedDT)
      end
      If RelatedDT.name = DT.name then
        Add Column of ForeignKey to Model(DT) for Self Reference
      end
      Push RelatedDT into RelatedDTs
    else
      If ST.qualified_cardinality = 1 then
        Add Column of ST.name to Model(DT)
      else ST.max_qualified_cardinality = 1 then
        Add Column of ST.name to Model(DT)
      else
        developNewModel(ST)
        Model(DT) has “One-to-many” relation with Model(ST)
      end
    end
  end

  If DT has ParentDT then
    Add Column of ForeignKey to Model(DT) for Model(ParentDT)
    If there are/is other ParentDT then
      Add Column of ForeignKey to Model(DT) for Model(ParentDT)
    end
  end
end
```

図 38　developRecordSet()アルゴリズム

6. 支援システムの実現

5 章では、支援システムを使ってどの様な支援を行うか、またその方法について説明した。本章では、まずメタデータ作成ツールの開発に用いる Ruby on Rails と、ひな形として生成する Rails Application Template を紹介する。その上で、5 章で説明した支援 1 を行う支援システム ToDsp と、支援 2 を行う支援システム ToRat について説明する。

6.1. Ruby on Rails

本研究では、メタデータ記述規則の設計手法としてメタデータ作成ツールの開発を通じて、メタデータ記述規則の大まかな設計を行うと述べた。一見、これは大変手間のかかる作業にも感じられる。しかし近年、開発を容易に行うための Web アプリケーションフレームワークが普及したことで、単純な Web アプリケーションであればすぐに開発することが出来る様になった。そのため本研究では、メタデータ作成ツールの開発方法として Ruby on Rails を用いることを想定している。

Ruby on Rails[13]とは、プログラミング言語 Ruby で書かれたオープンソースの Web アプリケーションフレームワークである。「同じことを繰り返すな」、「設定より規約」という原則を持ち、決まりに従ったコードを書くことで、Web アプリケーションの開発を行い易くしている。例えば、

```
$ rails new blog
```

というコマンドを入力すると、blog というディレクトリが生成され、その中には、表 4 の様な構成でアプリケーションの開発に必要なファイルやディレクトリが生成される。更にこのディレクトリ内で、

```
$ rails generate scaffold Post title:string text:text
$ rake db:migrate
$ rails server
```

というコマンドを実行し、Web ブラウザで URL にアクセスすると図 39 の様なフォームからタイトルとテキストを入力し、データを作成、更新、削除することが出来るようになる。そのため、短いサイクルでソフトウェアを反復的に開発していく必要があるアジャイル開発モデルには、Ruby on Rails は非常に有効なツールであると言える。

Ruby on Rails はモデル（Model）、ビュー（View）、コントローラ（Controller）の 3 つで構成される MVC と呼ばれるアーキテクチャで編成されている。モデルはアプリケーションのデータベースへの操作を行うための処理を行う。ビューはアプリケーションのユーザ

[13] http://rubyonrails.org/

インタフェースに関わる処理を行う。コントローラはモデルとビューを繋ぐための処理を行う。支援システムでは、とりわけモデルに記述するプログラムを重点的に取り扱う。

また、Ruby on Rails を用いたアプリケーション開発をより効率的に行うためのアイテムとして、Rails Application Template[14](RAT)がある。RAT とは Ruby on Rails の Template API に沿って書かれた Ruby のファイルである。アプリケーションを開発する際に毎回行っている行程などを RAT として記述しておくことで、何度も同じコマンドやコードを書く行為を省くことが出来る。支援システムでは、メタデータ記述規則の情報を基に、メタデータ作成ツールのひな形として RAT を生成する。これにより、コマンドを数回打ち込むだけで簡単にメタデータ作成ツールを開発することが可能となる。

表 4　Ruby on Rails で生成されるディレクトリやファイルの一覧

File/Directory	説明
app/	アプリケーションのコントローラやモデル、ビュー等についてのディレクトリを含む
bin/	アプリケーションを起動させるためのスクリプト等を含む
config/	アプリケーションの実行時のルール、ルーティング、データベース等の設定
config.ru	アプリケーションの起動のために使われるサーバのための設定
db/	データベースのスキーマ等についてのディレクトリ、ファイルを含む
Gemfile Gemfile.lock	アプリケーションに必要な gem（Ruby のライブラリ）依存が何かを指定する
lib/	拡張モジュール
log/	アプリケーションのログファイルを含む
public	ありのまま世界中に公開されるディレクトリ。静的ファイル等を含む
Rakefile	タスクを読み込みコマンドラインから実行出来る様に配置されているファイル
README.rdoc	アプリケーションのための簡単な取扱説明書
test/	Unit test や fixture、その他テスト様の仕組みについてのファイルを含む
vendor/	サードパーティのコードを配置する。主に、RubyGems や Rails のソースコードを含む
tmp/	一時的なファイルを含む

[14] http://guides.rubyonrails.org/rails_application_templates.html

図 39　Ruby on Rails によるアプリケーションの入力画面の例

6.2. 支援システムの開発とその利用方法

本研究では ToDsp と ToRat という 2 つの支援システムを開発した構成した。ToDsp は 5 章で説明した支援 1 を、ToRat は支援 2 を行うための機能を持つ。本説では、それぞれのシステムについて詳しく説明する。

6.2.1. ToDsp：グラフ構造の構築とメタデータ語彙の探索支援

ToDsp は Ruby on Rails で開発されたアプリケーション（以後、Rails アプリケーションと呼ぶ）が持つモデルに関するファイルを読み込み、メタデータ記述規則のひな形を生成するシステムである。操作は、全てコマンドライン上で行う。このアプリケーションを実行することで、ユーザであるメタデータスキーマ設計者は Ruby on Rails で開発されたメタデータ作成ツール上のモデルの関係を基に、メタデータ記述規則のひな形を作ることが出来る。

ToDsp では、まず入力値として、ユーザから Rails アプリケーションのディレクトリへのパスと、OWL-DSP の生成に必要な URI を受け取る。そして、パスの情報から Rails アプリケーション内で生成される以下のファイルを読み込む。

- db/schema.rb
- app/models ディレクトリに含まれる全てのファイル

図 40 は schema.rb の記述例、図 41 は models の中にあるファイルの記述例である。schema.rb は Ruby on Rails によって自動的に生成され、models のファイルは Ruby on Rails が自動的に生成したものをユーザが Ruby on Rails の規約に従って編集する。

schema.rb からは、どの様なモデル（テーブル）があり、それぞれ何のカラムを持つかを読み込む。app/models 内のファイルからは、それぞれのモデル持つ関連やカラムについての制約を読み込む。全てのモデルの情報を読み込むと、5.1.1 項で説明した構築パターンに沿って、RDB スキーマからグラフ構造を構築し、Description Template と Statement Template を作成していく。このときユーザは構築パターンによってモデルを、URI を持つリソースにするか、空白ノードにするか、または関連しているリソースのプロパティとするかという問い合わせに答える必要がある。更に、クラスやプロパティを 5.1.2 項で説明した方法で探索する。このときユーザは、LOV で用意されているカテゴリを利用して、メタデータ語彙の候補を絞り込むことが出来る。図 42 は ToDsp の実行画面である。赤い矢印部分は、ToDsp からの問い合わせで、緑の矢印部分はユーザの回答である。

メタデータ語彙の検索まで終了すると、出力ファイルとして RDF/XML で記述された OWL-DSP が生成される。出力された OWL-DSP には、コメント部分にそれぞれのメタデータ語彙の検索結果が、それぞれの Description Template や Statement Template に用いるメタデータ語彙の候補として LOD Popularity が大きい順番に記述されており（LOD Popularity が 0 の場合は結果に含まない）、ユーザはそれらを参考にメタデータ語彙を選択し、OWL-DSP を完成させる。ToDsp の出力結果の例は、付録 A に載せている。

また ToDsp では、4.2 節で説明した手法に沿って反復的にメタデータ記述規則の設計を行うことを踏まえて、入力ファイルとして既存の DSP も読み込む事が出来る様にした。これによって、既存の DSP に同じ名前を持つ Description Template や Statement Template が含まれている場合は、メタデータ語彙を新しい DSP に引き継ぐことが可能となっている。そのためユーザは、2 回目のメタデータ記述規則の設計からは、最初から新たなメタデータ記述規則を最初から作り直すのではなく、変更された部分のみを修正すれば良い。

```
ActiveRecord::Schema.define(version: 20140109050131) do

  create_table "books", force: true do |t|
    t.string     "book_uri"
    t.string     "title"
    t.string     "author_name"
    t.datetime "created_at"
    t.datetime "updated_at"
  end
end
```

図 40　db/schema.rb の記述例

```
class Book < ActiveRecord::Base
  has_one :author
  validates :book_uri, :presence => true, :uniqueness => true
  validates_presence_of :title
end
```

図 41　app/models にあるファイルの例（book.rb の場合）

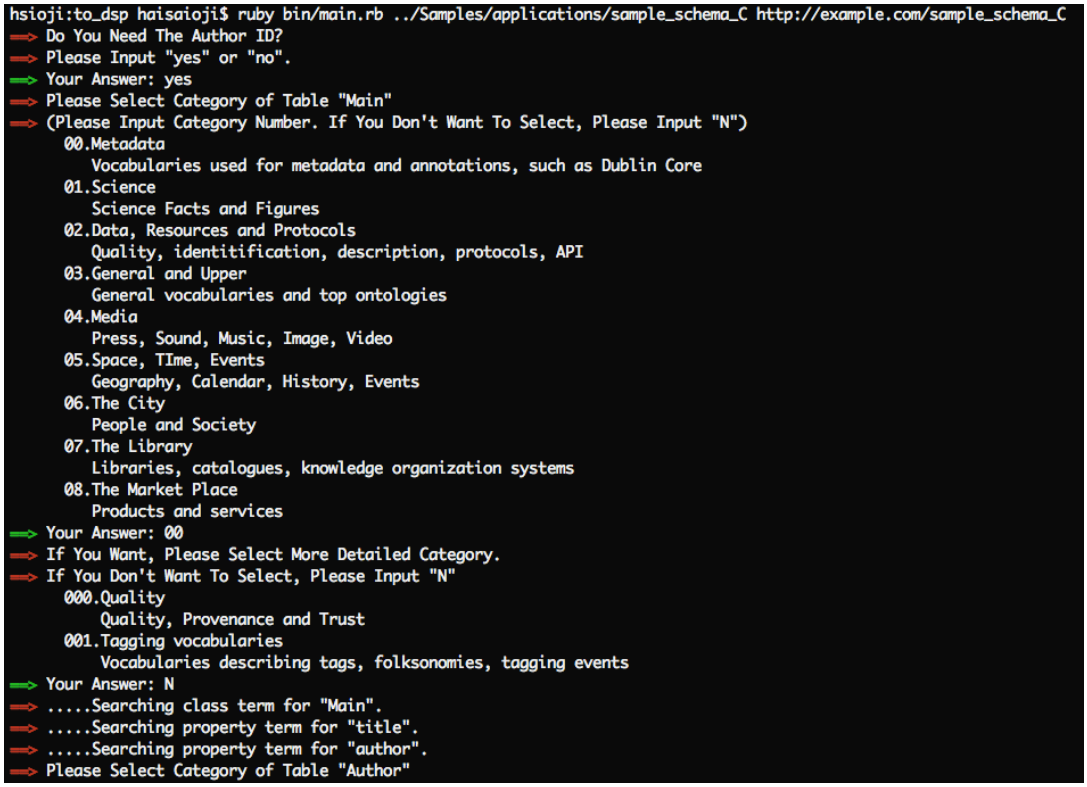


図 42　ToDsp の実行画面

6.2.2. ToRat：メタデータ作成ツールの DB スキーマ構築支援

ToRat は、RDF/XML 形式の OWL-DSP で表現されたメタデータ記述規則を読み込み、メタデータ作成ツールを開発するためのひな形となる RAT を生成するシステムである。ToDsp と同様に、操作は全てコマンドライン上で行う。このシステムを実行することで、メタデータスキーマ設計者はメタデータ記述規則を基に、メタデータ作成ツールのひな形を作ることが出来る。

ToRat では、まず入力値として、RDF/XML 形式の OWL-DSP で表現されたメタデータ記述規則のファイルと、OWL-DSP を読み込むために必要な URI を受け取る。それらを読み込むと、5.2 節で説明した構築パターンに沿って RDF のグラフ構造から DB スキーマを構築し、モデルやその関連を作るための Ruby のコードを RAT 上に記述していく。なお、

1 対 1 関連で空白ノードを持つ場合は 2 つの構築パターンが考えられたが、本システムは図 32 の Pattern1 に従って新たなモデルを生成している。

次にユーザは生成された RAT を使って、以下のコマンドで新たにメタデータ作成ツールを生成する。(application_name はメタデータ作成ツールの名前を、app_template.rb は Rails Application Template のファイル名をそれぞれ入力する)

```
$ rails new application_name -m app_template.rb
```

すると、application_name という新たなメタデータ作成ツールが生成され、

```
$ rails server
```

を実行するとメタデータ作成ツールを起動することが出来る。メタデータ作成ツールの URL にアクセスすると、メタデータスキーマ設計者は、メタデータ記述規則に沿ってデータを入力し作成することが出来る。また、更新、削除も可能である。図 43、図 44 は生成されたメタデータ作成ツールの表示画面の例である。データの入力フォームの生成は、Active Scaffold[15]という Rails のための Ruby ライブラリを利用し、モデルの関連から動的にフォームを生成している。また、入力したデータは RDB に保存されるが、図 45 の様に RDF ファイルとして出力することも可能である。そのため、ユーザはメタデータを試作するだけでなく、更にメタデータに対して SPARQL で検索を行うなどといった試用を行うことでメタデータスキーマが要求要件を満たしているかを判断することが出来る。

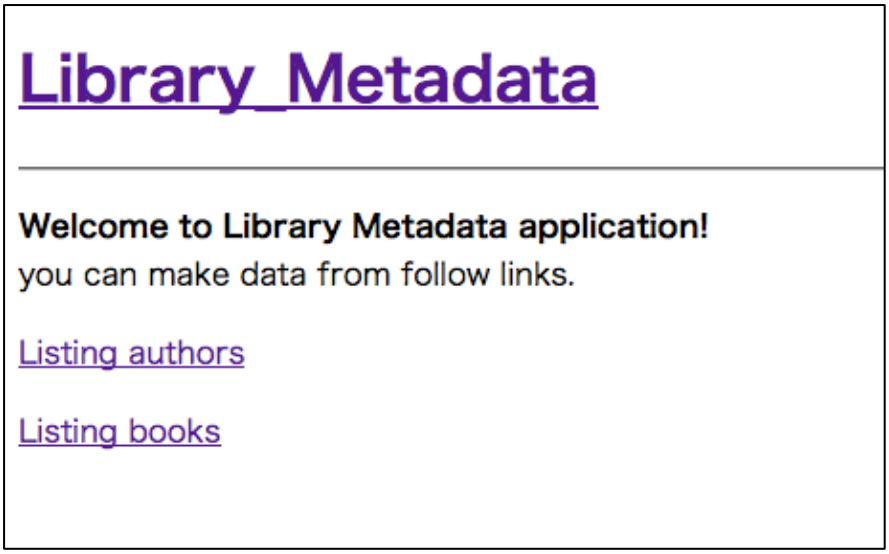


図 43　生成されたメタデータ作成ツールのトップ画面例

[15] https://github.com/activescaffold/active_scaffold

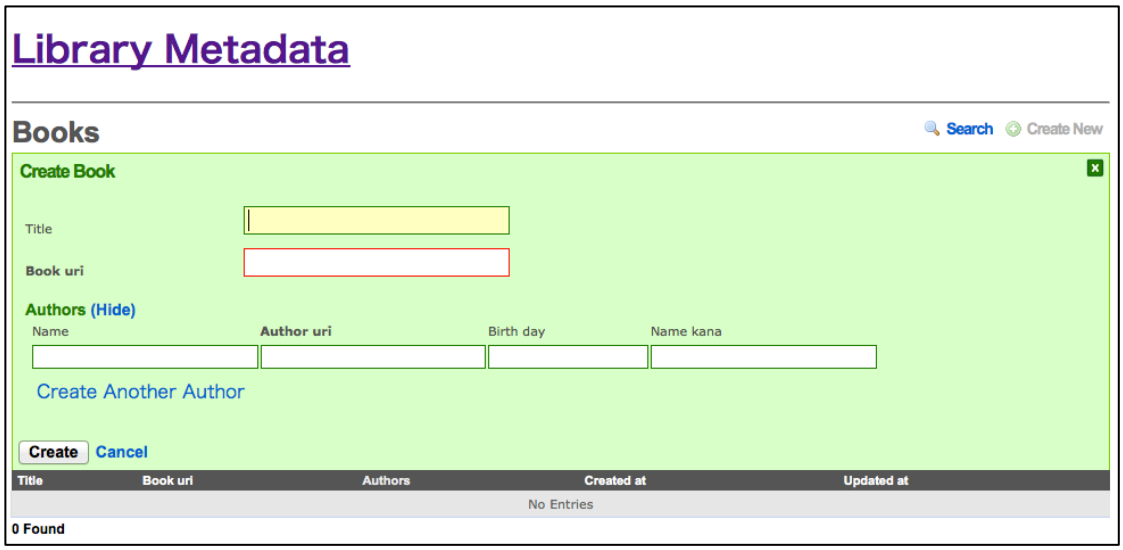


図 44　生成されたメタデータ作成ツールの入力フォーム例

```
−<rdf:RDF>
  −<dcterms:BibliographicResource rdf:about="http://example.com/book/01">
      <dcterms:title>坊ちゃん</dcterms:title>
    −<dc:creator>
      −<foaf:Person rdf:about="http://example.com/person/01">
          <foaf:name>夏目漱石</foaf:name>
          <ndl:transcription>なつめそうせき</ndl:transcription>
          <foaf:birthday>1967-02-09</foaf:birthday>
        </foaf:Person>
      </dc:creator>
    </dcterms:BibliographicResource>
  −<dcterms:BibliographicResource rdf:about="http://example.com/book/02">
      <dcterms:title>山月記</dcterms:title>
    −<dc:creator>
      −<foaf:Person rdf:about="http://example.com/person/02">
          <foaf:name>中島敦</foaf:name>
          <ndl:transcription>なかじまあつし</ndl:transcription>
          <foaf:birthday>1909-05-05</foaf:birthday>
        </foaf:Person>
      </dc:creator>
    </dcterms:BibliographicResource>
</rdf:RDF>
```

図 45　メタデータ作成ツールからの RDF 出力例

7. 支援システムの評価・考察

本章では、5 章と 6 章で説明した本研究で開発した支援システムに対する評価と考察について述べる。本研究では、ToDsp が実際にどの様な DB スキーマに対して RDF のグラフ構造を構築することが可能であるか、また ToRat がどの様なグラフ構造に対して DB スキーマの構築を行うことが出来るかを調査した。本節では、その調査方法と結果を述べた上で、それらについて考察を行った。また 7.2 節では、5.1.2 項で説明した手法を用いてメタデータ語彙の探索を行い、どのような結果を得られたかを調査した。得られた結果に対する考察を行った。

7.1. RDF のグラフ構造と RDB スキーマの構築に対する評価・考察

本研究では、開発した支援システム ToDsp と ToRat がどの様な RDF のグラフ構造や RDB スキーマのパターンに対して対応可能かを調査した。本節では調査の手順と結果、またそれらに対する考察について述べる。

本研究では、まず表 5 に挙げられるサンプル A からサンプル Q までの 17 個のメタデータ記述規則のサンプルを用意し、それらを入力値として支援システム ToRat を実行した。表 7 は ToRat の実行結果である。

まず、用意したメタデータ記述規則のサンプルについて説明する。5 章でも述べた様に、RDF のグラフ構造ではリソースは 1 対 1、1 対多、多対多という関係で関連し合っている。また、それらのリソースは、互いに関連せず独立しているもの、木構造の形で関連し合うもの、閉路を作って関連し合うものなどに区分される。ここで、サンプル A、B は関連を全く持たないもの、サンプル C、D、E、F、H、I、K は木構造となるもの、サンプル G はそれら両方を含むもの、サンプル J は一つのリソースに対し、2 つの異なるリソースが関連するものである。サンプル L は閉路を含む。サンプル M、N は互いに多対多で関連し合うもの、サンプル O は 1 つのリソースから同じクラスのリソースに対して 2 つのプロパティで関連するもの、サンプル P、Q は自己参照を持つものである。付録 B にこれらを図示したものを載せた。表 6 はメタデータ記述規則のサンプルを関連の種類と Description Template の数で分類したものである。なお、空白となっている Description Template の数が 2 つで互いに関連を持たない場合は、サンプル G がその条件を満たしている。また、自己参照を多対多関連で持つ場合、多対多関連を 2 つ以上持つ場合については、現在、本システムが対応していないことが自明となっている。

表 5　メタデータ記述規則のサンプル一覧

サンプル名	説明	調査対象
A	Description Template A は、他の Description Template と関連を持たず、全ての Statement Template の最大記述回数が 1 である	Description Template が何も関連を持たない場合の動作
B	Description Template A は、他の Description Template と関連を持たず、最大記述回数が N の Statement Template を 1 つ持つ	Statement Template の最大記述回数が N の場合の動作
C	Description Template A は、URI を持たない Description Template B（空白ノード）と 1 対 1 で関連を持ち、全ての Statement Template の最大記述回数は 1 である。	URI を持たない Description Template（空白ノード）と 1 対 1 で関連を持った場合の動作
D	Description Template A は、URI を持たない Description Template B（空白ノード）と 1 対多で関連を持ち、残り全ての Statement Template の最大記述回数は 1 である。	URI を持たない Description Template（空白ノード）と 1 対多で関連を持った場合の動作
E	Description Template A は、Description Template B と 1 対 1 で関連を持ち、全ての Statement Template の最大記述回数は 1 である。	URI を持つ Description Template と 1 対 1 で関連を持った場合の動作
F	Description Template A は、Description Template B と 1 対多で関連を持ち、残り全ての Statement Template の最大記述回数は 1 である。	URI を持つ Description Template と 1 対多で関連を持った場合の動作
G	Description Template A は、Description Template B と 1 対多で関連を持ち、全ての Statement Template の最大記述回数は N である。また、Description Template A、B とは関連を持たない Description Template C が 1 つ存在する。	互いに関連を持たない複数の Description Template が存在した場合の動作
H	Description Template A は、Description Template B と 1 対多で関連を持ち、Description Template B は更に Description Template C と 1 対多で関連を持つ。Description Template A、B の残りの Statement Template の最大記述回数は 1 であり、Description Template C の残りの Statement Template の最大記述回数は N である。	Description Template が深さ 3 の木構造の形で、1 対多で関連している場合の動作
I	Description Template A は、Description Template B と 1 対 1 で関連を持ち、Description Template B は更に Description Template C と 1 対 1 で関連を持つ。Description Template A、B の残りの Statement Template の最大記述回数は N であり、Description Template C の残りの Statement Template の最大記述回数は 1 である。	Description Template が深さ 3 の木構造の形で、1 対 1 で関連している場合の動作

J	Description Template A と Description Template B は共に 1 対多で Description Template C と関連を持つ。Description Template A、B、C の残りの Statement Template の最大記述回数は N である。	2 つの Description Template が同時に共通の Description Template を参照している場合の動作
K	Description Template A は、Description Template B と 1 対 1 で関連を持ち、Description Template C と 1 対多で関連を持つ。Description Template A、B、C の残りの Statement Template の最大記述回数は 1 である。	1 つの Description Template が同時に 2 つの Description Template を参照している場合の動作
L	Description Template A は、Description Template B と 1 対 1 で関連を持ち、Description Template B は Description Template C と 1 対多で関連を持つ。更に、Description Template C は Description Template A と 1 対多で関連を持つ。Description Template A、B、C の残りの Statement Template の最大記述回数は 1 である。	3 つの Description Template が関連し、閉路を作っている場合の動作
M	Description Template A は、Description Template B と多対多で関連を持つ。Description Template A、B の残りの Statement Template の最大記述回数は 1 である。	URI を持つ Description Template が多対多で関連を持った場合の動作
N	Description Template A は、Description Template B と 1 対多で関連を持ち、更に Description Template B と Description Template C は多対多で関連を持つ。Description Template A、B、C の残りの Statement Template の最大記述回数は 1 である。	URI を持つ Description Template が多対多で関連を持ち、更に他の関連を持つ場合の動作
O	Description Template A は、Description Template B と 2 つの Statement Template で 1 対多の関連を持つ。Description Template A、B の残りの Statement Template の最大記述回数は 1 である。	1 つの Description Template が同じ Description Template に対して、異なる Statement Template で 2 回以上参照している場合の動作
P	Description Template A は 1 対 1 の関連で自己参照の関連を持つ。残りの Statement Template の最大記述回数は 1 である。	1 対 1 の自己参照の関連を持つ場合の動作。
Q	Description Template A は 1 対多の関連で自己参照の関連を持つ。残りの Statement Template の最大記述回数は 1 である。	1 対多の自己参照の関連を持つ場合の動作。

表 6　メタデータ記述規則のサンプルの分類

		以下の関連を含む			
		関連なし	1 対 1	1 対多	多対多
Description Template の数	1	A,B	P	Q	
	2		C,E	D,F,G,O	M
	3	G	I,K,L	H,J,K,L	N

表 7 は、これらのサンプルを使って ToRat を実行した結果である。表 7 では、5 章で述べた構築パターン通りにグラフ構造から RDB スキーマを構築出来たものに○を、出来なかったものに×を付けている。ToRat を実行した結果として、5.2 節で説明した現在のアルゴリズムではサンプル L の様に 3 つ以上のリソースが循環して参照し、閉路を作る場合に無限ループが発生し対応出来ないことが分かった。これについては、循環が 1 周したところで再帰を止め、無限ループを回避する改善が必要であると考えられる。また、サンプル O の様に同じ Description Template を持つリソースに対して 2 つの異なるプロパティ（Statement Template）が伸びる場合にも対応出来ないことが分かった。この場合は、図 46 の様に、2 つのテーブルを構築するだけでなく、更にそれらを仲介する中間テーブル（図 46 ではテーブル Book-Author）を作成し、役割を記述するためのカラム（図 46 ではテーブル Book-Author のカラム role）を持たせることで、プロパティの違いを表現出来るように改善する必要がある。そのためこの様な関係を持つ場合、メタデータ作成ツールのユーザはデータを入力する度に、プロパティの代わりとなる役割も一緒に入力する必要がある。

表 7　ToRat の実行結果

サンプル名	実行結果	サンプル名	実行結果
A	○	J	○
B	○	K	○
C	○	L	×
D	○	M	○
E	○	N	○
F	○	O	×
G	○	P	○
H	○	Q	○
I	○		

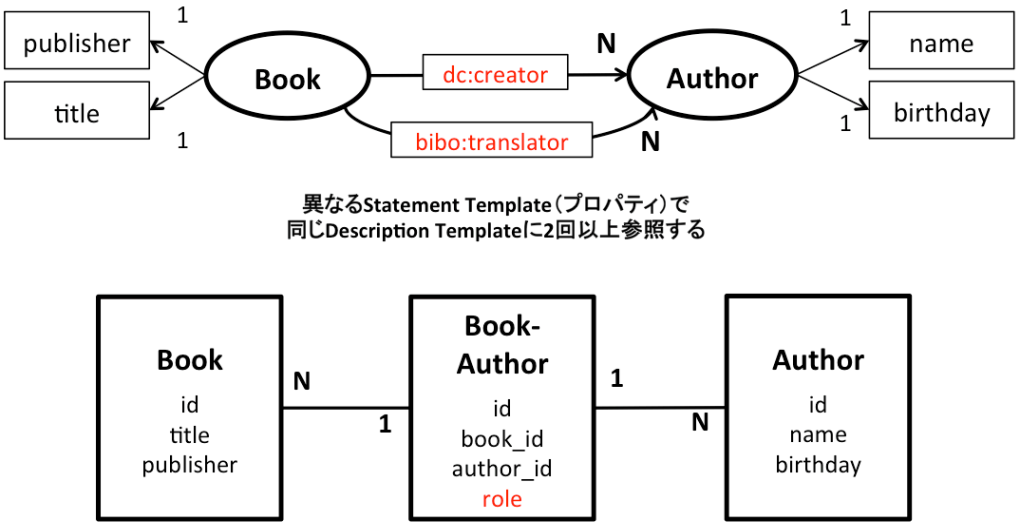


図 46　サンプル R のグラフ構造を表現した RDB スキーマ

また、表 8 に挙げる 18 個の Rails アプリケーションのサンプルを用意し、それらに対してアプリケーション ToDsp を実行した。また、そのうち 13 個は ToRat で生成した RAT を基に開発した Rails アプリケーションである。残る 5 つのサンプルについて説明する。サンプル 2 は、キー以外のカラムの数が 1 で、1 対 1 で関連するテーブルが存在する場合を調べる。サンプル 7 は多対多の関連を持つ際に、中間テーブルがキー以外のカラムを持つ場合を調べる。また、サンプル 8、9 は Active Record が持つ関連「has_one :through」、「has_and_belongs_to_many」を持つ場合を調べる。サンプル 16 はテーブルが循環して閉路を作って関連している場合を調べる。表 5 と同様に付録 B にこれらを図示したものを載せた。

表 8　Rails アプリケーションのサンプル一覧

サンプル名	説明	調査対象
1	テーブル A は他のテーブルと関連を持たない（サンプル A）	テーブルが関連を持たない場合の動作
2	テーブル A はテーブル B と 1 対 1 の関連を持ち、テーブル B のキー以外のカラムの数は 1 である。	テーブルが、キー以外のカラムの数が 1 であるテーブルと 1 対 1（has_one）の関連を持つ場合の動作
3	テーブル A はテーブル B と 1 対多の関連を持ち、テーブル B のキー以外のカラムの数は 1 である。（サンプル B）	テーブルが、キー以外のカラムの数が 1 であるテーブルと 1 対多（has_many）の関連を持つ場合の動作

4	テーブル A はテーブル B と 1 対 1 の関連を持ち、テーブル B のキー以外のカラムの数は N である。（サンプル C、E）	テーブルが、キー以外のカラムの数が N であるテーブルと 1 対 1（has_one）の関連を持つ場合の動作
5	テーブル A はテーブル B と 1 対多の関連を持ち、テーブル B のキー以外のカラムの数は N である。（サンプル D、F）	テーブルが、キー以外のカラムの数が N であるテーブルと 1 対多（has_many）の関連を持つ場合の動作
6	テーブル A はテーブル B とテーブル C を中間テーブルとして多対多（has_many through）の関連を持つ。テーブル C のキー以外のカラムの数は 0 である。（サンプル M）	テーブルが多対多（has_many through）の関連を持ち、中間テーブルのキー以外のカラムが 0 の場合の動作
7	テーブル A はテーブル B とテーブル C を中間テーブルとして多対多（has_many through）の関連を持つ。テーブル C のキー以外のカラムの数は 1 である。	テーブルが多対多（has_many through）の関連を持ち、中間テーブルのキー以外のカラムが 1 以上の場合の動作
8	テーブル A はテーブル C と 1 対 1（has_one through）の関連を持ち、テーブル B はテーブル C と 1 対 1（has_one）の関連を持つ。全てのテーブルのキー以外のカラムの数は 1 である。	テーブルが 1 対 1（has_one through）の関連を持つ場合の動作
9	テーブル A はテーブル B とテーブル C を中間テーブルとして多対多（has_and_belongs_to_many）の関連を持つ。テーブル C のキー以外のカラムの数は 0 である。	テーブルが多対多（has_and_belongs_to_many）の関連を持つ場合の動作
10	テーブル A はテーブル B、C と 1 対多の関連を持つ。テーブル B はテーブル D と 1 対多の関連を持つ。テーブル E はテーブル F、G と 1 体多の関連を持つ。テーブル C、D、F、G のキー以外のカラムの数は 1 である。（サンプル G）	関連を持たないテーブルが存在する場合の動作
11	テーブル A はテーブル B、C と 1 対多の関連を持つ。テーブル B はテーブル D と 1 対多の関連を持つ。テーブル C のキー以外のカラムは 1 である。（サンプル H）	テーブルが深さ 3 の木構造の形で、1 対多で関連している場合の動作
12	テーブル A はテーブル B と 1 対 1、テーブル C と 1 対多の関連を持つ。テーブル B はテーブル D と 1 対 1、テーブル E と 1 対多の関連を持つ。テーブル C、テーブル E のキー以外のカラムは 1 である。（サンプル I）	テーブルが深さ 3 の木構造の形で、1 対 1 で関連している場合の動作
13	テーブル A はテーブル B、C と 1 対多の関連を持つ。テーブル B はテーブル D と 1 体多の関連を持つ。テーブル E はテーブル B、F と 1 体多の関連を持つ。テーブル C、D、F のキー以外のカラムは 1 である。（サンプル J）	2 つのテーブルが同時に共通のテーブルを参照している場合の動作
14	テーブル A はテーブル B と 1 対多、テーブル C と 1 対 1 の関連を持つ。テーブル B、C のキー以外のカラムは N である。（サンプル K）	1 つのテーブルが同時に 2 つのテーブルを参照している場合の動作

15	テーブル A はテーブル B と 1 対多の関連を持つ。テーブル B はテーブル C と多対多（has_many through）の関連を持つ。（サンプル N）	テーブルが多対多の関連を持ち、更に別の関連を持つ場合の動作
16	テーブル A はテーブル B と 1 対多の関連を持つ。テーブル B はテーブル C と 1 対 1 の関連を持つ。テーブル C はテーブル A と 1 対多の関連を持つ。	3 つのテーブルが関連し、閉路を作っている場合の動作
17	テーブル A は 1 対 1 の自己参照を持つ。（サンプル P）	テーブルが 1 対 1 の自己参照を持つ場合の動作
18	テーブル A は 1 対多の自己参照を持つ。（サンプル Q）	テーブルが 1 対多の自己参照を持つ場合の動作

表 9　ToDsp の実行結果

サンプル名	実行結果	サンプル名	実行結果
1	○	10	○
2	○	11	○
3	○	12	○
4	○	13	○
5	○	14	○
6	○	15	○
7	○	16	×
8	○	17	○
9	○	18	○

表 9 はこれらの Rails アプリケーションのサンプルを ToDsp で実行した結果である。表 9 では、5 章で述べた変換通りに RDB スキーマからグラフ構造を生成したものに○を、出来なかったものに×を付けている。ここではサンプル 1 からサンプル 15、サンプル 17、サンプル 18 は上手く対応することが出来た。しかしながら、ToRat と同様にサンプル 16 の 3 つのテーブルが循環する閉路となっている場合は対応することが出来なかった。本アルゴリズムの MAIN 部分では、テーブルが他のテーブルに従属する関係を持っていた場合にテーブルの生成が重複しない用に、ループをスキップする様にしていた。しかしながら、この場合ではどのテーブルもいずれかのテーブルに従属する関係になってしまうため、Description Template を生成する作業に入ることが出来なかった。そのため、他のテーブルに従属する関係を持っていた場合でも、循環する関係を持っているかどうかを確認出来るように改善する必要がある。

以上の様に、本研究では ToRat と ToDsp を実行する調査を行うためにそれぞれ 17 個のメタデータ記述規則（DSP）のサンプルと、18 個の Rails アプリケーションを用意した。しかしながら、これらは全ての RDF のグラフ構造と RDB スキーマの表現パターンを網羅しているとは言えない。一方で、RDF のグラフ構造と RDB スキーマの表現パターンを完全に網羅することは非常に難しいと考えられる。そのため、今後アルゴリズムを改善していくと同時に、実際によく用いられる RDF のグラフ構造や RDB スキーマの表現パターンを調査・分析し、優先的に対応していく必要があると考えられる。

また、実際に ToRat によって生成されたメタデータ作成ツールを利用してメタデータを作成した。これにより、一度、あるリソースと関連付けて入力したリソースの値を、更に他のリソースと結び付ける場合があること、またその場合には現在のメタデータ作成ツールでは、その都度メタデータを入力し直す必要があるということが分かった。例えば、組織（Organization）とメンバー（Member）の関係を 1 対多で記述するとする。異なる組織に共通するメンバーが存在した場合、現在のメタデータ作成ツールでは図 47 の様にテーブル上のキーや外部キーが異なり、他の値が全く同じメタデータを再度入力する必要がある。これは、データの本質に沿った記述方法とは言えず、本来は多対多の関連にするべきであったと考えられる。現在の DB スキーマの構築パターンで、多対多の関連を結ぶことが出来るのは、URI を持つ 2 つのリソースが互いにプロパティを最大出現回数 N で結ぶ場合のみである。そのため、改めて構築パターンを見直し、URI を持つリソース同士が 1 つのプロパティで結ばれる場合にも、多対多関連で結ぶことが出来る様にする必要がある。

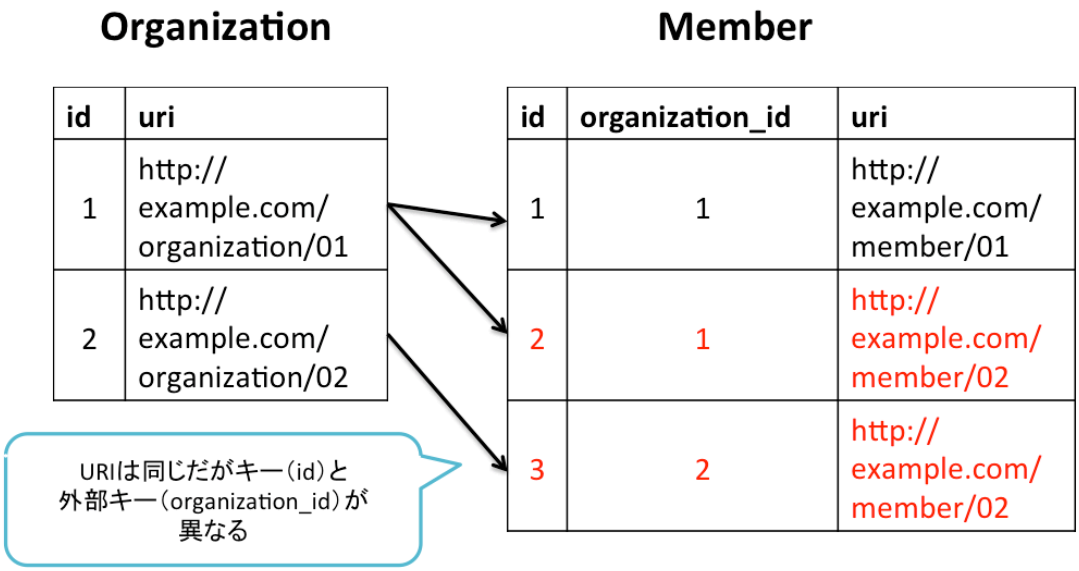


図 47　構築された DB スキーマに対する問題

7.2. メタデータ語彙の探索に対する評価・考察

本研究で開発した支援システム ToDsp では、テーブル名、カラム名を基に 5.1.2 項で説明した方法でメタデータ語彙の探索を行った。本節では、その検索結果に対する考察を述べる。ToDsp を使ってカラム「creator」、「author」、「writer」に対するプロパティを、カテゴリを絞らずに検索したところ、検索結果の数はそれぞれ 5 件、20 件、0 件であった。また、表 9 は「creator」、「author」の検索結果である。この結果から、同じ様な意味を持つ言葉同士であっても、どの言葉を検索キーワードとして用いるかによって検索結果は大きく異なることが分かった。そのため、今後の課題として検索キーワードによる格差をなくすために概念辞書のデータを持つ WordNet[16]などを利用して類義語、上位語による検索を可能にすることが考えられる。また、テーブル名から検索したクラス、カラム名から検索したプロパティそれぞれが持つ制約や、それぞれを実際に利用している LOD のデータを使って、条件を絞り込むことも考えられる。例えば、カラム名から検索したプロパティの制約で指定されたクラスを検索結果として出す、テーブル名から検索したクラスを持つリソースによく用いられるプロパティを候補として出すというものである。

表 10　「creator」、「author」の検索結果

creator	LOD popularity	author	LOD popularity
dcterms:creator	4781374	dcterms:contributor	6307520
dcterms:creator	4781374	dcterms:contributor	6307520
dce:creator	974116	mads:isIdentifiedByAuthority	5093133
dcndl:transcription	139606	schema:author	974556
foaf:maker	12754	dce:creator	974116
		rdag2:cataloguersNote	18188
		mrel:hnr	11091
		mrel:prf	1162
		mrel:aut	84
		mrel:aui	19

[16] http://wordnet.princeton.edu/

8. おわりに

本研究では、既存のメタデータ記述規則の設計プロセスについてまとめ、メタデータ記述規則の設計プロセスはメタデータの試作を通じて設計を繰り返すアジャイル開発モデルに基づくプロセスであると考えた。その上で、アジャイル開発モデルに基づくメタデータ記述規則の設計プロセスの問題点としてメタデータの試作に手間がかかること、それを改善するために必要なメタデータ作成ツールの開発にも手間がかかってしまうことを挙げた。また一方で、メタデータ記述規則の設計には RDF のグラフ構造の構築やメタデータ語彙の探索のために、RDF や LOD に関する専門的な知識や経験も求められるという問題点があった。そこで本研究では、メタデータ作成ツールの開発をメタデータ記述規則の設計プロセスに組み込むことで、アジャイル開発に基づいたメタデータ記述規則の設計プロセスをより効率的に行えるようにすることを目指した。

アジャイル開発に基づいたメタデータ記述規則の設計プロセスをより効率的に行うために、本研究では ToRat と ToDsp という 2 つの支援システムの開発を行った。ToDsp ではメタデータ作成ツールの DB スキーマの情報から、RDF のグラフ構造の構築や、メタデータ語彙の探索を行い、メタデータ記述規則のひな形の生成を行った。ToRat ではメタデータ記述規則の情報を基に、DB スキーマの構築を行い、メタデータ作成ツールを開発するためのひな形を生成する支援を行った。また、支援システムの評価を行うためにいくつかのサンプルとなるメタデータ記述規則（DSP）と Rails アプリケーションを用意し、RDF のグラフ構造と RDB スキーマの構築がどの様なパターンにまで対応出来るのかを調査した。この調査では、関連を持たない、もしくは木構造を持つ RDF のグラフ構造、DB スキーマには対応出来た。多対多を多く含む、もしくは循環した閉路を持つ RDF のグラフ構造、DB スキーマには対応出来なかった。また、ある Description Template から同じ Description Template に対して異なる Statement Template が結ばれている場合のグラフ構造にも対応出来なかった。そのため、本研究の支援システムは関連が少なく、木構造の様な単純な構造には対応することが出来たが、多対多の関連を多く含む、循環を含んだ閉路を持つ等の様な複雑な構造には対応出来ないことが分かった。また ToRat で生成したひな形を基に開発したメタデータ作成ツールを利用したところ、URI を持つリソース同士を結ぶ場合は、プロパティを互いに結んでいない場合でも多対多関連の DB スキーマを構築する必要があることが分かった。

今後これらに対応するための構築パターン、アルゴリズムの改善が求められる。それと同時に、既存のメタデータやデータベースで用いられる RDF のグラフ構造、DB スキーマを調査・分析し、よく用いられているものから優先的に構築パターンを増やしていく必要があると考えられる。加えて、メタデータ語彙の探索についても精度を挙げるために、概

念辞書などを用いた支援が求められる。今後の展望として、本支援システムを用いた提案手法が、実際のメタデータ記述規則の設計に対してどこまで有効であるか更なる検証を行なっていきたい。

9. 謝辞

本研究を進めるにあたり、指導教員の杉本重雄先生、副指導教員の永森光晴先生、本間維先輩、三原鉄也先輩には、大変多くのご指導を頂きました。本当に有難うございました。また日頃の議論を通じて、助言や励ましを頂いた杉本・永森研究室の同期、後輩の皆様にも深く感謝致します。

参考文献

[1]. Open Knowledge Foundation. “What is Open Data”. Open Data Handbook. http://opendatahandbook.org/en/what-is-open-data/, （参照 2014-01-01）

[2]. Tim Berners-Lee. “Linked Data – Design Issues”. 2006. http://www.w3.org/DesignIssues/LinkedData.html, （参照 2014-01-01）

[3]. “5 star Open Data”. http://5stardata.info/, （参照 2014-01-01）

[4]. Tim Berners-Lee. “W3 future directions”, Plenary at WWW Geneva 94. 1994. http://www.w3.org/Talks/WWW94Tim/, （参照 2014-01-10）

[5]. 神崎正英. セマンティック HTML/XHTML. 毎日コミュニケーションズ. 2009.

[6]. “RDF”. Semantic Web Standards. http://www.w3.org/RDF/, （参照 2014-01-01）

[7]. 神崎正英. セマンティック・ウェブのための RDF/OWL 入門. 森北出版. 2005.

[8]. Linked Data. http://linkeddata.org/, （参照 2014-01-01）

[9]. Mitsuharu Nagamori, Masahide Kanzaki, Naohisa Torigoshi, Shigeo Sugimoto. “Meta-Bridge: A Development of Metadata Information Infrastructure in Japan”. 2011. Proceedings of the DCMI International Conference on Dublin Core and Metadata Applications 2011, pp.63-68

[10]. メタデータ基盤協議会. 新 ICT 利活用サービス送出支援事業（電子出版の環境整備）メタデータ情報基盤構築事業報告書. 2010. http://www.mi3.or.jp/item/metaproj2010.pdf, （参照 2014-01-10）

[11]. “RDF Vocabulary Description Language 1.0: RDF Schema”. http://www.w3.org/TR/rdf-schema/, （参照 2014-01-01）

[12]. Mikael Nilsson. “Description Set Profiles: A constraint language for Dublin Core Application Profiles”. http://dublincore.org/documents/dc-dsp/, （参照 2014-01-01）

[13]. メタデータ基盤協議会. “メタデータ・スキーマ定義言語”. http://www.mi3.or.jp/item/A04.pdf, （参照 2014-01-01）

[14]. “OWL Web Ontology Language Overview”. http://www.w3.org/TR/owl-features/, （参照 2014-01-01）

[15]. Karen Coyle, Thomas Baker. “Guidelines for Dublin Core Application Profiles”. http://dublincore.org/documents/profile-guidelines/, （参照 2014-01-01）

[16]. Mikael Nilsson, Thomas Baker, Pete Johnston. “The Singapore Framework for Dublin Core Application Profiles”. http://dublincore.org/documents/2008/01/14/singapore-framework/, （参照 2014-01-01）

[17]. メタデータ基盤協議会. “メタデータ情報共有のためのガイドライン”. http://www.mi3.or.jp/item/A03.pdf, （参照 2014-01-01）

[18]. 江渡浩一郎. “11 章 エクストリームプログラミング”. パターン、Wiki、XP. 技術評論者. 2009, p.89-107.

[19]. Jonathan Rasmusson, 近藤修平, 角掛拓未, 西村直人, 角谷信太郎. アジャイルサムライ –達人開発者への道. 株式会社オーム社. 2011.

[20]. “アジャイルソフトウェア開発宣言”. http://www.agilemanifesto.org/iso/ja/, （参照 2012-01-01）

[21]. Mariana Curado Malta, Ana Alice Baptista. “A Method for the Development of Dublin Core Application Profiles (Me4DCAP V0.2): Detailed Description”. 2013. Proceedings of the DCMI International Conference on Dublin Core and Metadata Applications 2013, pp.90-103

[22]. Satya S. Sahoo, Wolfgang Halb, Sebastian Hellmann, Kingsley Idehen, Ted Thibodeau Jr, Sören Auer, Juan Sequeda, Ahmed Essat. “A Survey of Current Approaches for Mapping of Relational Databases to RDF”. W3C RDB2RDF Incubator Group Report. 2009.

[23]. Matthias Hert, Gerald Reif, Harald C. Gall. “A Comparison of RDB-to-RDF Mapping Language”. I-SEMANTICS 2011, Proceedings of the 7th International Conference on Semantic Systems. pp.25-32. 2011

[24]. Wondu Y. Mallede, Farhi Marir, Vassil T. Vassilev. “Algorithms for Mapping RDB Schema to RDF for Facilitating Access to Deep Web”. WEB 2013, The First International Conference on Building and Exploring Web Based Environments. 2013.

[25]. Tim Berners-Lee. “Relational Database and the Semantic Web (in the Design Issues)”. 1998. http://www.w3.org/DesignIssues/RDB-RDF.html, （参照 2014-01-07）

[26]. “SPARQL 1.1 Query Language”. 2013. http://www.w3.org/TR/sparql11-query/, （参照 2014-01-07）

[27]. Tsunagu Honma, Mitsuharu Nagamori, Shigeo Sugimoto. “Find and Combine Vocabularies to Design Metadata Application Profiles using Schema Registries and LOD Resources”. 2013. Proceedings of the DCMI International Conference on Dublin Core and Metadata Applications 2013, pp.104-114.

[28]. 西出頼継, 本間維, 永森光晴, 杉本重雄. 日本の Open Data 活用を目的としたデータセットのスキーマ分析とリンク関係の調査. 一般社団法人情報処理学会. 情報処理学

会研究報告. 情報学基礎研究会報告 2013-IFAT-112(4), pp.1-8, 2013.

付録 A

本研究で開発した支援システム ToDsp を実行して生成されたメタデータ記述規則(DSP)のファイルをここに掲載しておく。6.2.1 項でも述べた様に、ToDsp からは RDF/XML で記述された OWL-DSP のひな形が出力され、メタデータ語彙の検索結果はコメント部分（赤字部分）に記述されている。なお、ここで掲載したメタデータ記述規則は 7 章で述べた Rails アプリケーションのサンプル 5 の RDB スキーマから構築した。また、テーブル Author とそのカラムに対するメタデータ語彙の探索では、LOV カテゴリの大カテゴリ「The City」を使って検索結果を絞り込んだ。テーブル Book とそのカラムに対するメタデータ語彙の探索では、LOV カテゴリの大カテゴリ「Metadata」を使って検索結果を絞り込んだ。

```
<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:reg="http://purl.org/metainfo/terms/registry#"
xmlns:dsp="http://purl.org/metainfo/terms/dsp#"
xmlns:owl="http://www.w3.org/2002/07/owl#"
xmlns:dc="http://purl.org/dc/elements/1.1/"
xmlns:dcterms="http://purl.org/dc/terms/"
xmlns:skos="http://www.w3.org/2004/02/skos/core#"
xmlns:xl="http://www.w3.org/2008/05/skos-xl#"
xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
xmlns:foaf="http://xmlns.com/foaf/0.1/">
  <rdf:Description rdf:about="http://example.com/sample_application_5">
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#Ontology"/>
    <reg:created>2014-01-15</reg:created>
    <reg:title>sample_application_5</reg:title>
    <reg:primaryDescription
rdf:resource="http://example.com/sample_application_5#Author"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://example.com/sample_application_5#Author">
    <rdf:type
rdf:resource="http://purl.org/metainfo/terms/dsp#DescriptionTemplate"/>
    <dsp:resourceClass rdf:resource=""/>
    <!-- ### How About This Terms For "Author"? ###
    * http://xmlns.com/foaf/0.1/Person (foaf:Person) Popularity:2320027
    * http://xmlns.com/foaf/0.1/Agent (foaf:Agent) Popularity:695736
 -->
    <reg:idField>AuthorID</reg:idField>
    <rdfs:label>Author</rdfs:label>
    <rdfs:subClassOf
rdf:resource="http://example.com/sample_application_5#Author-Name"/>
    <rdfs:subClassOf
rdf:resource="http://example.com/sample_application_5#Author-NameKana"/>
    <rdfs:subClassOf
rdf:resource="http://example.com/sample_application_5#Author-BirthDay"/>
    <reg:statementOrder>name,name_kana,birth_day</reg:statementOrder>
```

```
  </rdf:Description>
  <rdf:Description
rdf:about="http://example.com/sample_application_5#Author-Name">
    <rdf:type
rdf:resource="http://purl.org/metainfo/terms/dsp#StatementTemplate"/>
    <owl:minQualifiedCardinality>0</owl:minQualifiedCardinality>
    <owl:maxQualifiedCardinality>1</owl:maxQualifiedCardinality>
    <owl:onProperty rdf:resource=""/>
    <!-- ### How About This Terms For "name"? ###
    * http://xmlns.com/foaf/0.1/name (foaf:name) Popularity:3317798
    * http://xmlns.com/foaf/0.1/givenname (foaf:givenname) Popularity:185937
    * http://xmlns.com/foaf/0.1/family_name (foaf:family_name) Popularity:179956
    * http://xmlns.com/foaf/0.1/givenName (foaf:givenName) Popularity:106834
    * http://xmlns.com/foaf/0.1/familyName (foaf:familyName) Popularity:103561
    * http://xmlns.com/foaf/0.1/firstName (foaf:firstName) Popularity:20404
    * http://xmlns.com/foaf/0.1/surname (foaf:surname) Popularity:17884
    * http://xmlns.com/foaf/0.1/mbox_sha1sum (foaf:mbox_sha1sum) Popularity:14967
    * http://xmlns.com/foaf/0.1/lastName (foaf:lastName) Popularity:216
    * http://xmlns.com/foaf/0.1/nick (foaf:nick) Popularity:76
    * http://xmlns.com/foaf/0.1/accountName (foaf:accountName) Popularity:61
    * http://usefulinc.com/ns/doap#name (doap:name) Popularity:7
    * http://purl.org/vocab/aiiso/schema#name (aiiso:#name) Popularity:1
 -->
    <rdfs:label>name</rdfs:label>
    <owl:onDataRange
rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
  </rdf:Description>
  <rdf:Description
rdf:about="http://example.com/sample_application_5#Author-NameKana">
    <rdf:type
rdf:resource="http://purl.org/metainfo/terms/dsp#StatementTemplate"/>
    <owl:minQualifiedCardinality>0</owl:minQualifiedCardinality>
    <owl:maxQualifiedCardinality>1</owl:maxQualifiedCardinality>
    <owl:onProperty rdf:resource=""/>
    <!-- ### How About This Terms For "name_kana"? ###
 -->
    <rdfs:label>name_kana</rdfs:label>
    <owl:onDataRange
rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
  </rdf:Description>
  <rdf:Description
rdf:about="http://example.com/sample_application_5#Author-BirthDay">
    <rdf:type
rdf:resource="http://purl.org/metainfo/terms/dsp#StatementTemplate"/>
    <owl:minQualifiedCardinality>0</owl:minQualifiedCardinality>
    <owl:maxQualifiedCardinality>1</owl:maxQualifiedCardinality>
    <owl:onProperty rdf:resource=""/>
    <!-- ### How About This Terms For "birth_day"? ###
    * http://xmlns.com/foaf/0.1/birthday (foaf:birthday) Popularity:1
 -->
    <rdfs:label>birth_day</rdfs:label>
    <owl:onDataRange rdf:resource="http://www.w3.org/2001/XMLSchema#date"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://example.com/sample_application_5#Book">
```

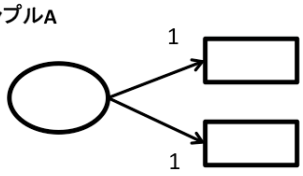
```
    <rdf:type
rdf:resource="http://purl.org/metainfo/terms/dsp#DescriptionTemplate"/>
    <dsp:resourceClass rdf:resource=""/>
    <!-- ### How About This Terms For "Book"? ###
    *                              http://purl.org/dc/terms/BibliographicResource
(dcterms:BibliographicResource) Popularity:7357973
 -->
    <reg:idField>BookID</reg:idField>
    <rdfs:label>Book</rdfs:label>
    <rdfs:subClassOf
rdf:resource="http://example.com/sample_application_5#Book-Title"/>
    <rdfs:subClassOf
rdf:resource="http://example.com/sample_application_5#Book-Author"/>
    <reg:statementOrder>title,author</reg:statementOrder>
  </rdf:Description>
  <rdf:Description
rdf:about="http://example.com/sample_application_5#Book-Title">
    <rdf:type
rdf:resource="http://purl.org/metainfo/terms/dsp#StatementTemplate"/>
    <owl:minQualifiedCardinality>0</owl:minQualifiedCardinality>
    <owl:maxQualifiedCardinality>1</owl:maxQualifiedCardinality>
    <owl:onProperty rdf:resource=""/>
    <!-- ### How About This Terms For "title"? ###
    * http://purl.org/dc/terms/title (dcterms:title) Popularity:13420023
    * http://purl.org/dc/elements/1.1/title (dce:title) Popularity:7626720
    *                                       http://purl.org/dc/elements/1.1/title
(http://purl.org/dc/elements/1.1/title) Popularity:7626720
    *         http://purl.org/dc/terms/alternative        (dcterms:alternative)
Popularity:735177
    *      http://ndl.go.jp/dcndl/terms/transcription     (dcndl:transcription)
Popularity:139606
 -->
    <rdfs:label>title</rdfs:label>
    <owl:onDataRange
rdf:resource="http://www.w3.org/2000/01/rdf-schema#Literal"/>
  </rdf:Description>
  <rdf:Description
rdf:about="http://example.com/sample_application_5#Book-Author">
    <rdf:type
rdf:resource="http://purl.org/metainfo/terms/dsp#StatementTemplate"/>
    <owl:minQualifiedCardinality>0</owl:minQualifiedCardinality>
    <owl:onProperty rdf:resource=""/>
    <!-- ### How About This Terms For "author"? ###
    *         http://purl.org/dc/terms/contributor        (dcterms:contributor)
Popularity:6307520
    *                      http://www.loc.gov/mads/rdf/v1#isIdentifiedByAuthority
(mads:#isIdentifiedByAuthority) Popularity:5093133
    * http://purl.org/dc/elements/1.1/creator (dce:creator) Popularity:974116
    *                            http://www.loc.gov/mads/rdf/v1#authoritativeLabel
(mads:#authoritativeLabel) Popularity:2
    *                     http://www.loc.gov/mads/rdf/v1#isMemberOfMADSCollection
(mads:#isMemberOfMADSCollection) Popularity:2
    *     http://www.loc.gov/mads/rdf/v1#adminMetadata     (mads:#adminMetadata)
Popularity:1
```

```
   *                          http://www.loc.gov/mads/rdf/v1#hasBroaderAuthority
(mads:#hasBroaderAuthority) Popularity:1
 -->
    <rdfs:label>author</rdfs:label>
    <owl:onClass
rdf:resource="http://example.com/sample_application_5#Author"/>
  </rdf:Description>
</rdf:RDF>
```

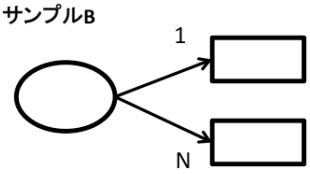
付録 B

本稿の 7 章の実験に用いたサンプルを分かり易く図示したものをここに示す。なお、N は N>1 の自然数とする。

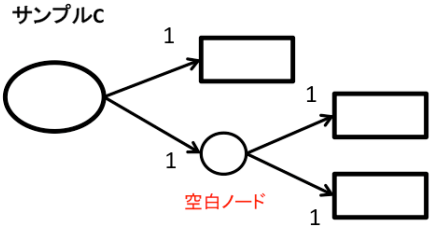
サンプルA



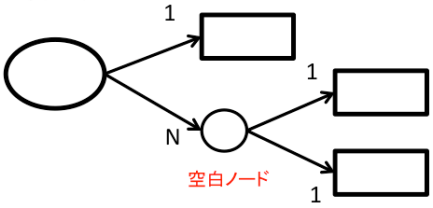
サンプルB



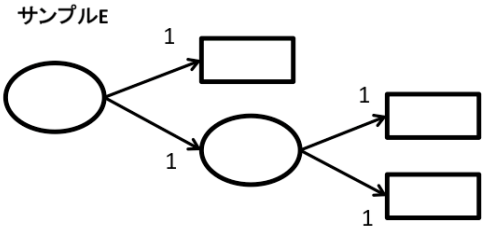
サンプルC



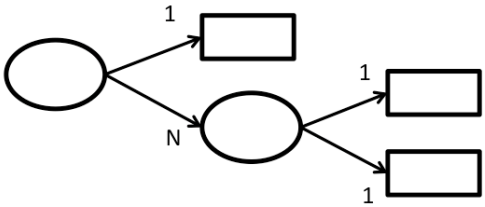
サンプルD



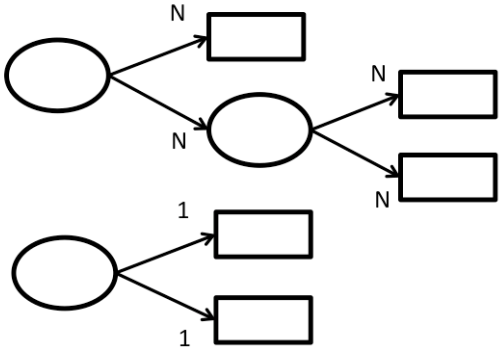
サンプルE



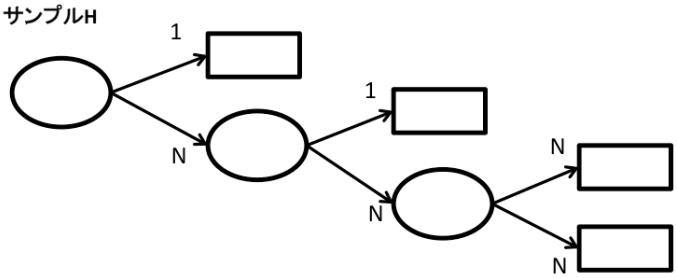
サンプルF



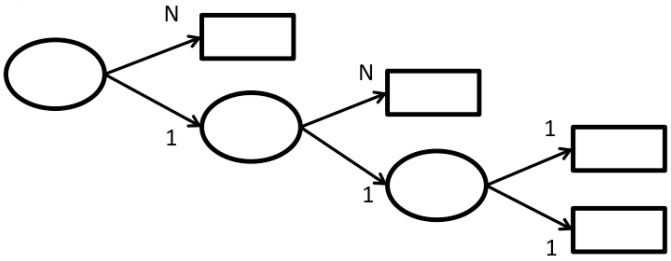
サンプルG



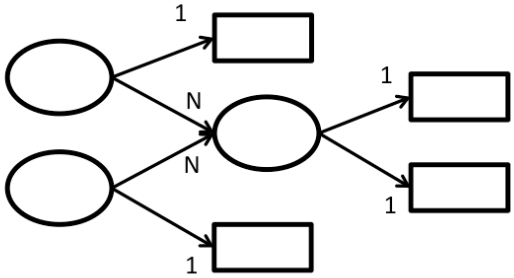
サンプルH



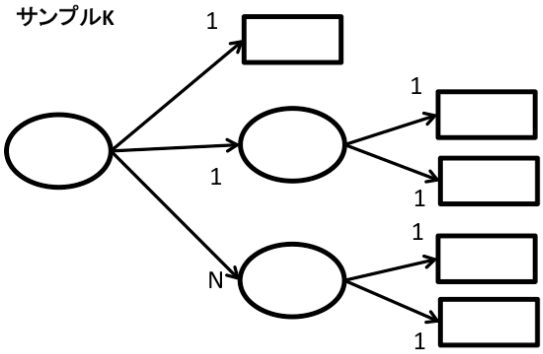
サンプルI



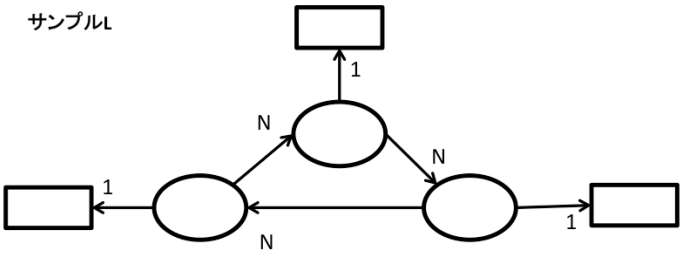
サンプルJ



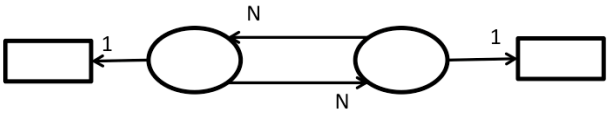
サンプルK



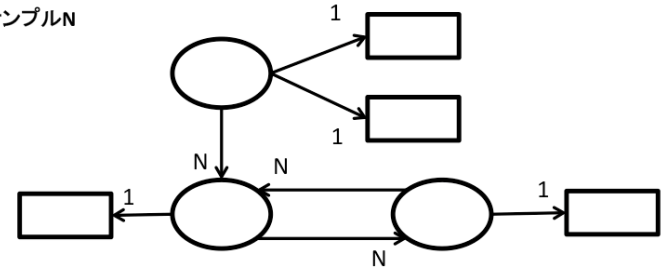
サンプルL



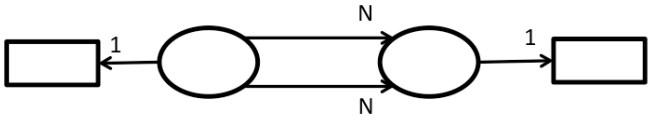
サンプルM



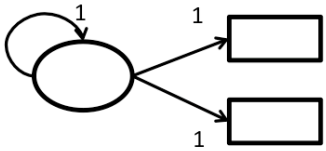
サンプルN



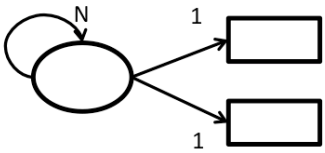
サンプルO



サンプルP

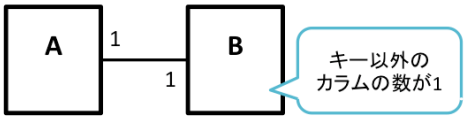


サンプルQ

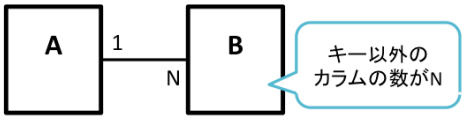


サンプル1

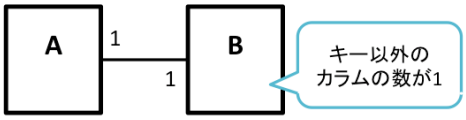
サンプル2



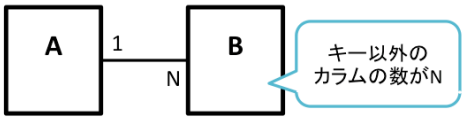
サンプル3



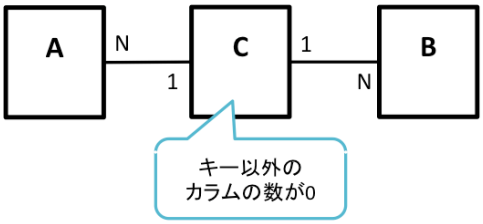
サンプル4



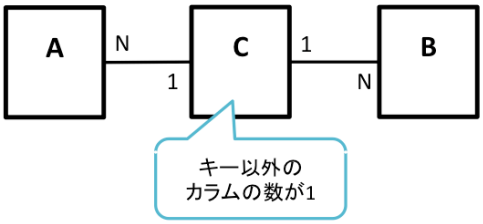
サンプル5



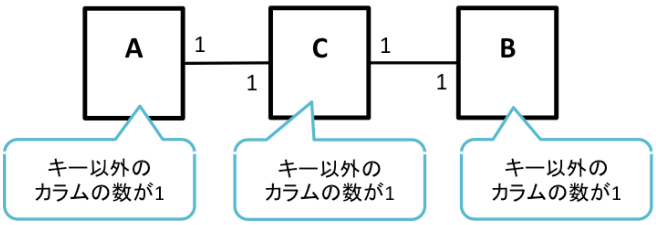
サンプル6,9



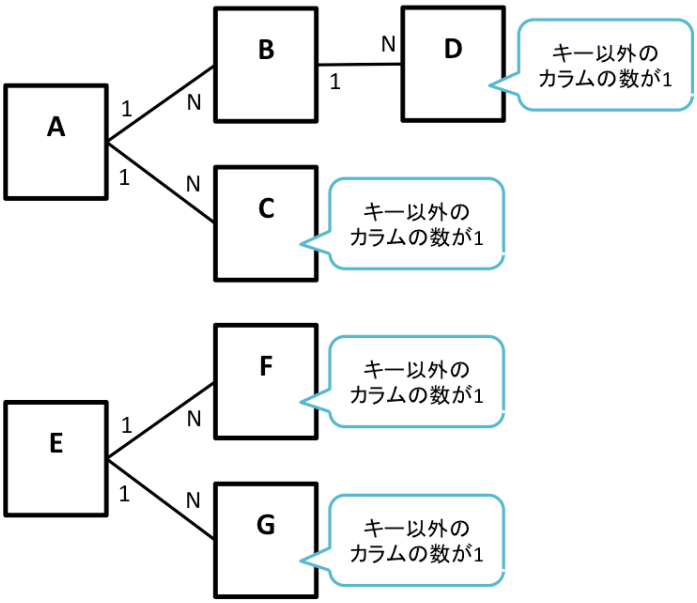
サンプル7



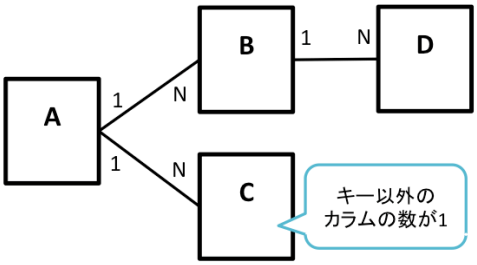
サンプル8



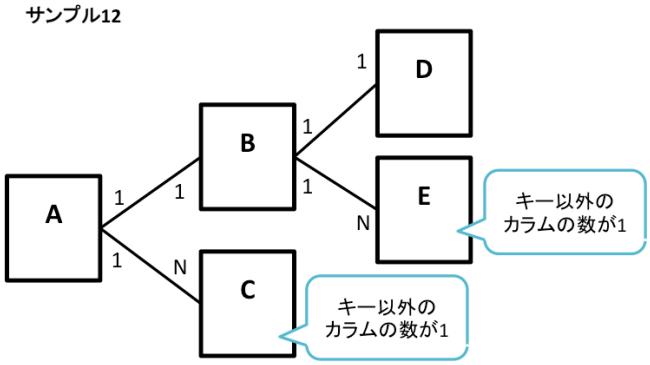
サンプル10



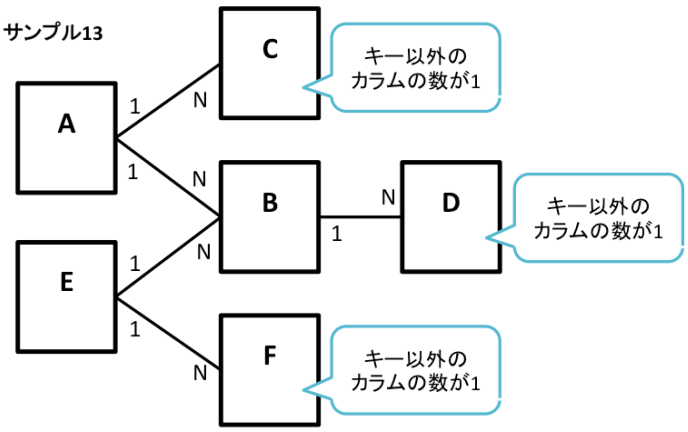
サンプル11



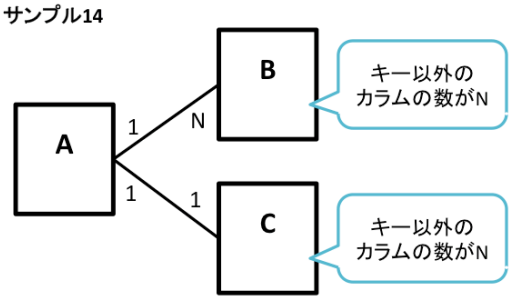
サンプル12



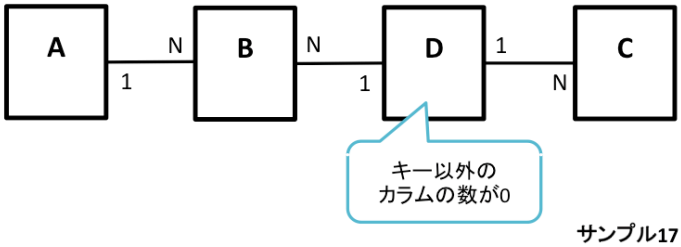
サンプル13



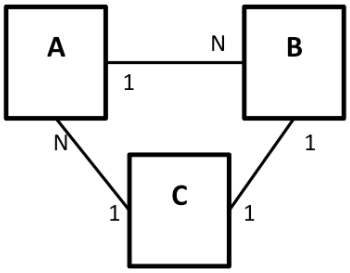
サンプル14



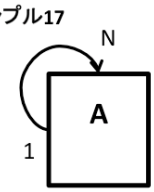
サンプル15



サンプル16



サンプル17



サンプル18